NAVAL POSTGRADUATE SCHOOL
Monterey, California

BE 2123

THESIS

DESIGN, IMPLEMENTATION AND EVALUATION OF AN ABSTRACT PROGRAMMING AND COMMUNICATIONS INTERFACE FOR A NETWORK OF TRANSPUTERS

by

Gregory R. Bryant

June 1988

Thesis Advisor: Uno R. Kodres

Approved for public release; distribution is unlimited

ECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

a. REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; Distribution is unlimited
b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
. PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School — 6b. OFFICE SYMBOL (If applicable) Code 52	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
c. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
a. NAME OF FUNDING/SPONSORING ORGANIZATION — 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

1. TITLE (Include Security Classification)
esign, Implementation and Evaluation of an Abstract Programming and Communications nterface for a Network of Transputers.

2. PERSONAL AUTHOR(S)
BRYANT, GREGORY R.

3a. TYPE OF REPORT Master's Thesis	13b. TIME COVERED FROM ______ TO ______	14. DATE OF REPORT (Year, Month, Day) 1988 June	15 PAGE COUNT 176

6. SUPPLEMENTARY NOTATION
The views in expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

7. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Distributed computing; Transputer; OCCAM; Network; Event Counts and Sequencers; Communicating Sequential Processes

9. ABSTRACT (Continue on reverse if necessary and identify by block number)
This thesis presents the design, implementation and evaluation of two abstracted programming and communication interfaces for developing distributed programs on a network of Transputers. One interface uses a shared memory model for interprocess communication and synchronization. The other interface uses a message passing model for communication and synchronization. The programming interfaces allow development of distributed programs that are independent of the physical configuration of a network. This thesis also presents an evaluation of Transputer performance with a particular emphasis on the interaction of computation and inter-Transputer communication.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Professor Uno R. Kodres	22b TELEPHONE (Include Area Code) (408) 646-2197 — 22c. OFFICE SYMBOL Code 52Kr

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

Design, Implementation and Evaluation of an Abstract Programming and Communication Interface for a Network of Transputers

by

Gregory R. Bryant
Lieutenant Commander, United States Navy
B.S.E.E., University of New Mexico, 1975

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1988

ABSTRACT

This thesis presents the design, implementation and evaluation of two abstracted programming and communication interfaces for developing distributed programs on a network of Transputers. One interface uses a shared memory model for interprocess communication and synchronization. The other interface uses a message passing model for communication and synchronization. The programming interfaces allow development of distributed programs that are independent of the physical configuration of a network. This thesis also presents an evaluation of Transputer performance with a particular emphasis on the interaction of computation and inter-Transputer communication.

THESIS DISCLAIMER

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While every effort has been made within the time available to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

Many terms used in this thesis are registered trademarks of commercial products. Rather than attempting to cite each individual occurrence of a trademark, all registered trademarks appearing in this thesis are listed below the firm holding the trademark:

INMOS Limited, Bristol, United Kingdom:
- Transputer
- OCCAM
- IMS T414
- IMS T800
- Transputer Development System (TDS)

Relational Technology Inc., Alameda, California:
- Ingres

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

DEDICATION

I dedicate this thesis to my wife, Kathy, who gave me confidence and encouragement and to my children, Betsey and Rusty, who were patient and understanding.

I. INTRODUCTION

A. BACKGROUND

The Aegis Modeling Project in the Computer Science Department at the Naval Postgraduate School is engaged in researching advanced computer architectures for potential future application aboard naval ships. Currently, this research is centered on the application and evaluation of distributed computing architectures. A distributed architecture is particularly suited for shipboard applications. Shipboard locations at which processing capabilities are required are physically distributed, yet processors at all locations must cooperate to monitor and control a ship's sensors and systems. Such an architecture can also provide the necessary characteristics of high performance, fault tolerance, and extensibility.

One emphasis of the current research has been to investigate systems that are composed of relatively low cost, "off-the-shelf" components. The single chip microprocessor is such a component. A range of microprocessor-based distributed multicomputer systems are, therefore, being evaluated. One such system already developed and evaluated uses clusters of microprocessor-based single-board computers interconnected by a hierarchical bus structure [Ga86]. A distributed system architecture now being evaluated is based on the single chip microprocessor known as the Transputer.

B. THE TRANSPUTER

The Transputer is a single chip microprocessor that has been specifically designed to function as a computing element in a distributed multicomputer system. The name "Transputer" is an amalgam of the words **trans**istor and com**puter**. As the transistor was a building block for large and varied electronic circuits, the Transputer is intended by the manufacturer to be an analogous building block for distributed computing systems.

To facilitate the use of the Transputer as an element in a distributed system, the Transputer implements the concept of Communicating Sequential Processes [Ho79]. Communicating Sequential Processes is a paradigm which defines and describes the interaction of programs that execute in parallel (as is the case in a distributed system).

C. ABSTRACT PROGRAMMING INTERFACES

Program development for a network of Transputers is currently closely tied to the particular physical configuration of a given network. The physical configuration of a network must be considered early and throughout the software design process. In certain cases, the configuration can actually dictate aspects of software design. This thesis investigates isolating the software designer from this physical configuration through the use of an abstract programming interface.

D. AVAILABLE HARDWARE AND SOFTWARE

The Transputer hardware available to the Aegis Modeling Project is varied. The hardware includes Transputer interface cards for personal computers, Transputer-based serial interface and color graphics interface cards, and cards with multiple Transputers. Aspects of this hardware that pertain to portions of this thesis are discused where applicable. A complete description of the hardware is available elsewhere [In86].

Programs included in this thesis were developed using the Transputer Development System(TDS) [In87a] and the OCCAM programming language [PoMa87]. OCCAM is a high-level block-structured language which includes constructs based on CSP to support programming in a distributed environment. An assembler and Pascal and C compilers for the Transputer are also available.

E. THESIS ORGANIZATION

Chapter II describes the Transputer and the basic concept of Communicating Sequential Processes.

This thesis investigates the basic performance characteristics of a Transputer as an element in a network of Transputers. Chapter III describes testing that was performed and documents the results of this testing.

Chapter IV and Chapter V describe and evaluate two different prototype abstract programming interfaces developed for a network of Transputers. One interface is based on a virtual globally shared memory. The other is based on message passing.

Chapter VI presents the conclusions reached as a result of developing, using, and evaluating the programming interfaces. Recommendations for further research are also provided in Chapter VI.

II. THE TRANSPUTER

A. OVERVIEW

Central to the to the Transputer is the concept of Communicating Sequential Processes (CSP) [Ho79]. The Transputer is, in fact, a hardware implementation of this concept. The programming language OCCAM, which is the primary language used for programming the Transputer, is based on this concept. A summary of CSP is presented in this chapter.

To effectively evaluate and use the Transputer requires an understanding of how the Transputer implements the concept of CSP. In addition, in many cases this thesis refers to Transputer architectural features and details. This chapter, therefore, also presents a brief description of Transputer architecture.

B. COMMUNICATING SEQUENTIAL PROCESSES

Communicating Sequential Processes (CSP) is one model for concurrent or parallel programming. In CSP, a program is composed of processes. A process consists of a list of commands or instructions that are to be executed in sequence. The different processes within a program are combined and specified to be executed in sequence or in parallel or in some sequential/parallel combination. The data spaces for any processes executed in parallel are constrained to be disjoint.

This requirement for parallel processes to have disjoint data spaces precludes using shared memory to communicate between the processes. To provide for necessary process-to-process communication, CSP instead utilizes message passing. Messages are passed between any pair of parallel processes via synchronous, unbuffered, point-to-point communications channels connected between the processes. These communication channels also provide the means for synchronizing processes. To communicate between two processes, one process must include an instruction for performing an output to the other process and the other process must include a corresponding input instruction. Both processes must be at that point in their instruction execution where the communication is specified to occur (If one process reaches this point first, it waits for the other process to reach its point of communication). The communication is then performed. Since the communication only occurs when both process are at their points of communication, the processes are synchronized at these points. In addition, CSP includes constructs for program control and sequencing and for conditional selection between multiple communications.

Figure 2.1 depicts a set of processes (represented as circles) interconnected by point-to-point communications links (represented as directed lines). This set of processes would operate independently and in parallel on a continuous stream of input values to produce a continuous stream of output values.

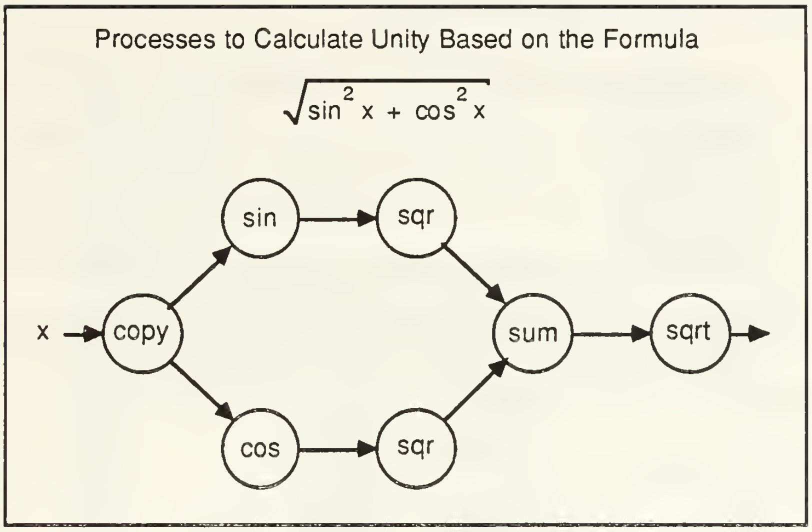

Processes to Calculate Unity Based on the Formula
$\sqrt{\sin^2 x + \cos^2 x}$
x
copy
sin
sqr
cos
sqr
sum
sqrt

Figure 2.1. **Process Representation Example**

Groups of processes may be logically aggregated to form larger, more abstracted process constructs. Figure 2.2 shows one possible abstraction of a subset of the processes shown in Figure 2.1.

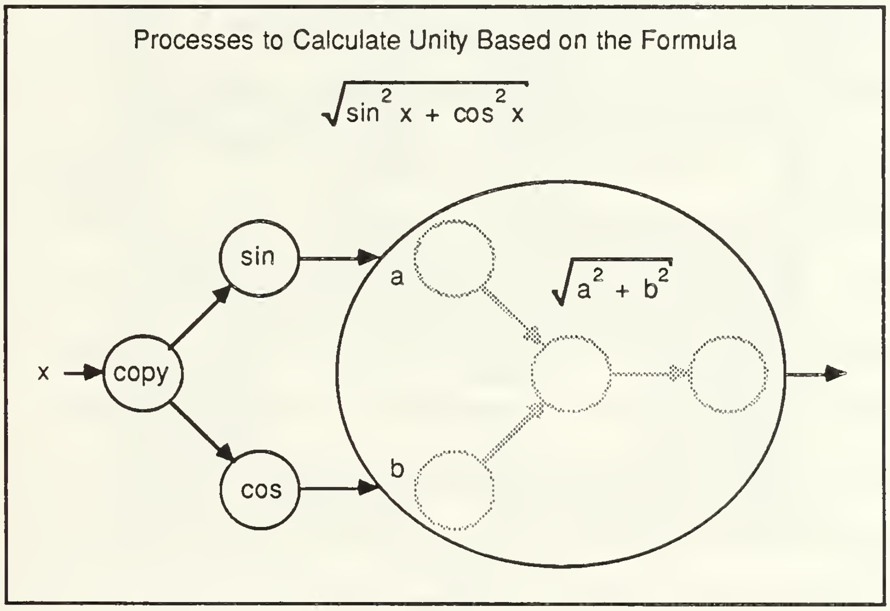

Processes to Calculate Unity Based on the Formula
$\sqrt{\sin^2 x + \cos^2 x}$
x
copy
sin
cos
a
b
$\sqrt{a^2 + b^2}$

Figure 2.2. **Process Abstraction Example**

C. TRANSPUTER ARCHITECTURE

A block diagram of a typical Transputer is shown in Figure 2.3. This figure depicts the major architectural components a Transputer. The following sections give a brief description of each of these components. In particular, these sections point out some architectural aspects of the Transputer that can influence the performance of programs written for the Transputer.

Several versions of the Transputer are currently available. This thesis considers only Transputer types T414 and T800. For this reason, the following sections describe the features of these Transputer types. A complete description of all currently available Transputers can be found elsewhere [In87b].

1. Processor

In general, the processor consists of a 32-bit integer arithmetic unit and a set of 32-bit registers. Figure 2.4 shows a block diagram of the processor. The **I** or instruction register points to the next instruction to be executed in a process. The **W** or workspace register is a pointer to the beginning of an area in memory that is the data space for a process. Together, these two registers can be thought of as representing the instructions and the data for a single process. The **A**, **B**, and **C** registers form a push-down evaluation stack. In general, all operations are performed on or using the values in these registers. For example, the load and store operations load and store the value of the **A** register. The add operation adds the values of the **A** and **B** registers leaving the result in the **A** register.

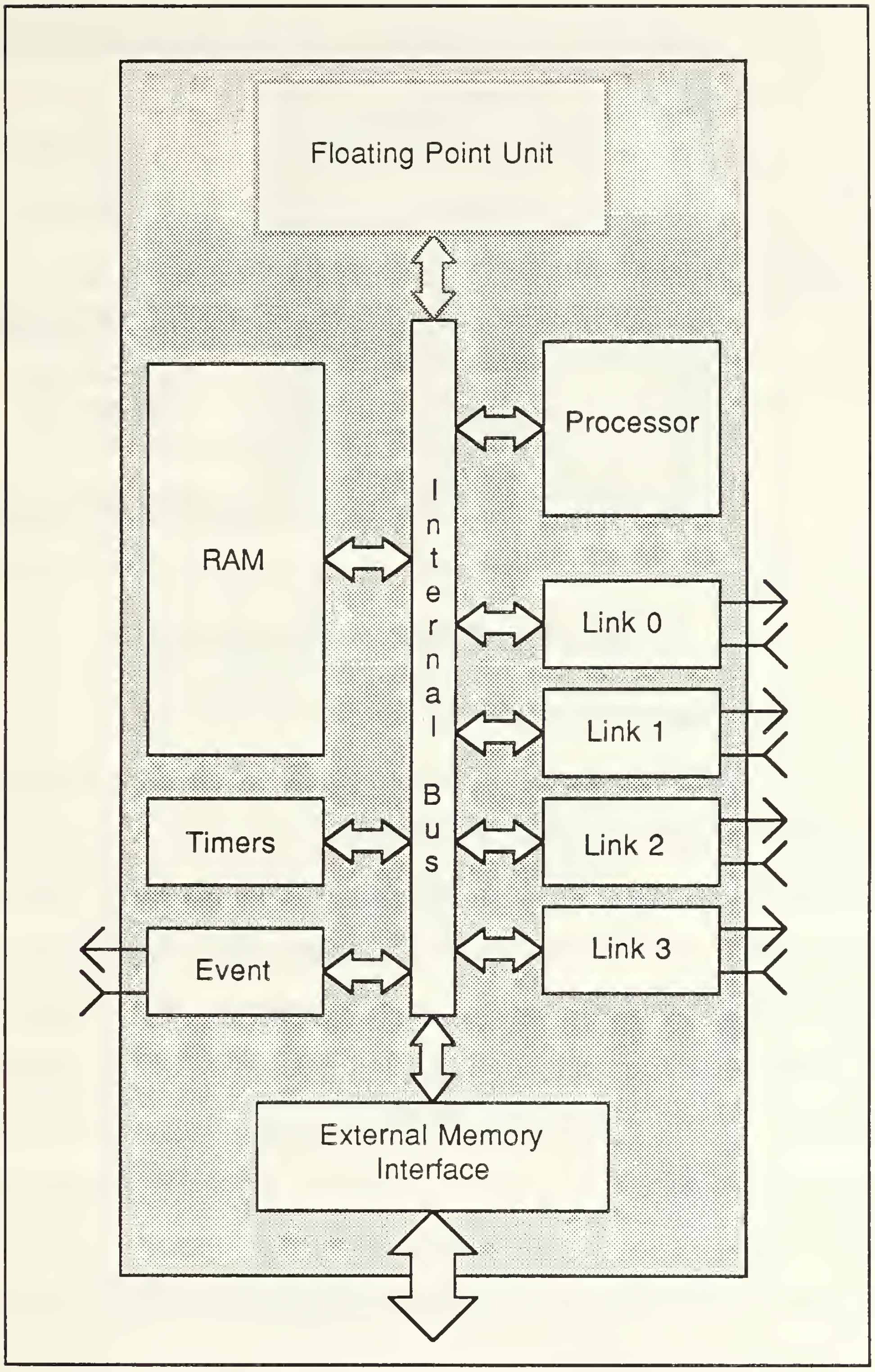


Figure 2.3. **Transputer Functional Block Diagram**

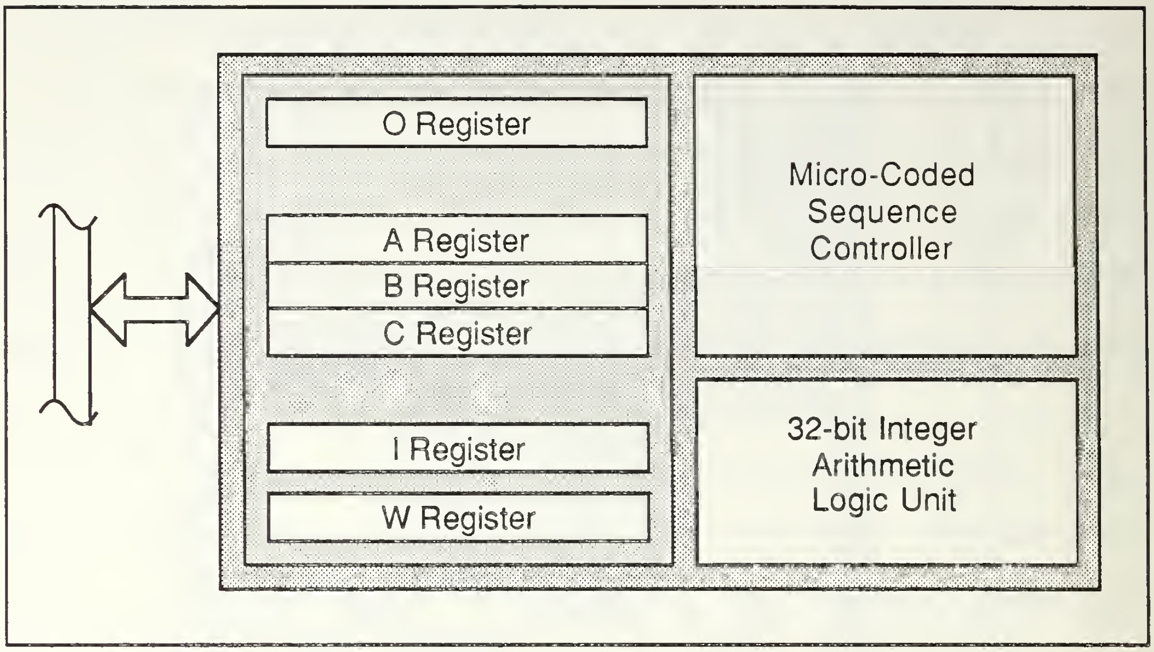
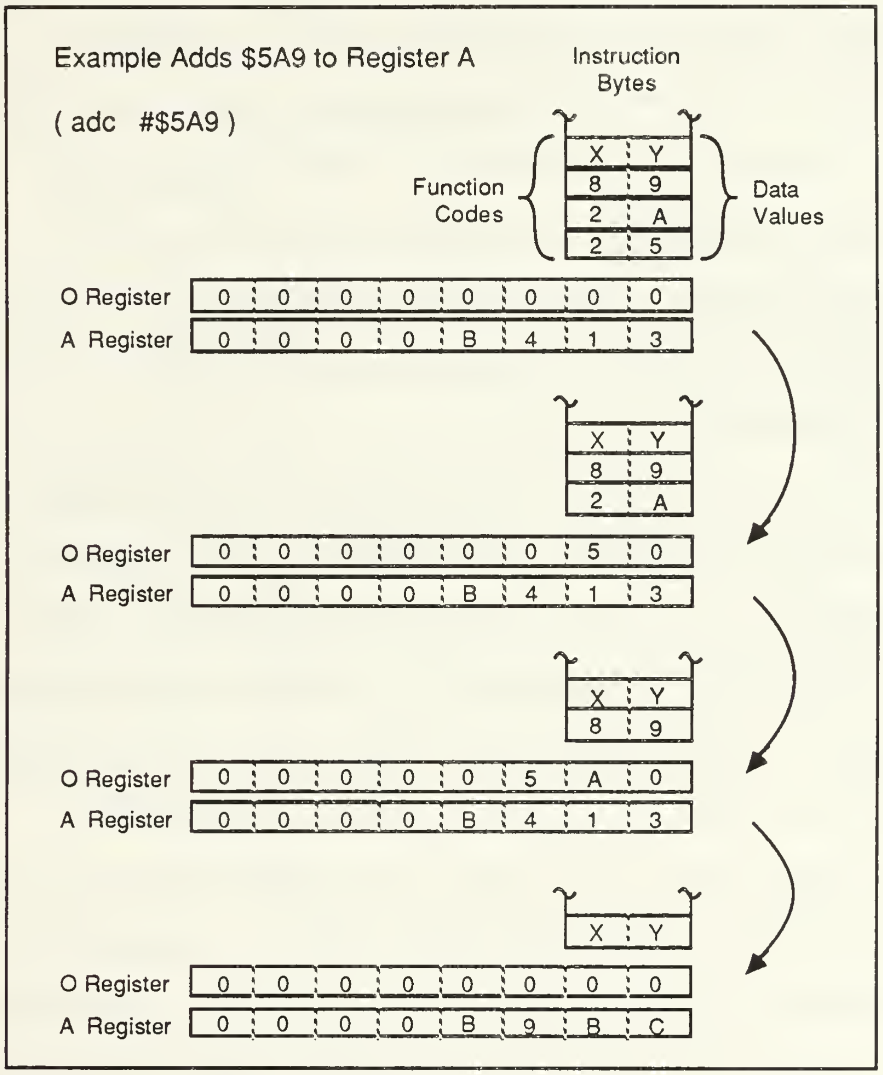

O Register
A Register
B Register
C Register
I Register
W Register
Micro-Coded Sequence Controller
32-bit Integer Arithmetic Logic Unit

Figure 2.4. **Processor Block Diagram**

a. Instruction Set

Each byte in a program can be viewed as having a four-bit high-order half and a four-bit low-order half. The low-order half of an instruction byte contains a data value. The high-order half contains a function code. To execute such an instruction, the data value half of the instruction byte is loaded into the operand register and the function encoded in the other half of the instruction byte is performed. At this point, it would appear that this instruction format limits operand values to the range of 0 to 15 and that only 16 different functions could be performed. A means is provided, however, for representing larger data values and for encoding a greater number of functions. The **O** or operand register is used to form operands and multi-byte instructions from a sequence bytes. For data values represented by more than four bits, special function codes cause individual four bit "pieces" of the data value to be extracted from a series of instruction bytes and accumulated in the operand register. The last function code in this series of instruction bytes will actually operate on the accumulated data value. Figure 2.5 shows an example of this operand formation process.



Example Adds $5A9 to Register A
(adc #$5A9)
Instruction Bytes
Function Codes
Data Values
X Y
8 9
2 A
2 5
O Register 0 0 0 0 0 0 0 0
A Register 0 0 0 0 B 4 1 3
O Register 0 0 0 0 0 0 5 0
A Register 0 0 0 0 B 4 1 3
O Register 0 0 0 0 0 5 A 0
A Register 0 0 0 0 B 4 1 3
O Register 0 0 0 0 0 0 0 0
A Register 0 0 0 0 B 9 B C

Figure 2.5. **Example of Operand Register Operation**

To encode functions with value representations greater than four bits, the value representing such a function is loaded into the operand register in the same manner as if it were data. A special function code then causes the value in the operand register to be executed as an instruction. Using this instruction formatting scheme, the instruction set has been optimized so that the most frequently used operations are encoded using only a single byte. Measurements show that about 70% of the instructions actually executed in a typical program are, in fact, encoded using only a single byte [In87c]. Having most instructions represented as single bytes not only reduces the memory requirements for program code but tends to improve program performance. This is because, since fewer bytes are fetched per instruction executed, fewer memory accesses will be required to execute the program.

This method of encoding instructions has other effects on processor performance. Each byte of a multi-byte instruction takes one processor clock cycle to load into the operand register for assembly of the instruction or its operand. Because of this, instructions with a greater byte length take more time to assemble before they can be executed. In most cases, the length of an instruction is fixed. However, in some cases, the length of an instruction is dependent on the value or location of the instruction's operand. Since these factors can be controlled by the programmer, it is useful to be aware of these types of instructions. For example, data values located in the first 16 locations above the base of the processor workspace require only one instruction byte to be accessed. Data values located elsewhere require additional instruction bytes, which increases the time required to access these data values. Because of this, frequently accessed data values should be located closest to the base of the workspace. Also, loading a constant takes one instruction byte for each four bits of the constant's length. This means that loading a 32-bit constant takes eight processor clock cycles. If such a constant is to be used repeatedly, it is often more efficient to name and store the constant as a data value and use that data value instead.

b. Concurrency

A single Transputer directly supports running concurrent processes. These processes may be either of two priority levels: high or low. To facilitate implementation of concurrency and prioritization, the Transputer has a micro-coded process scheduler. The operations performed by this scheduler can be examined based on whether a process is active or inactive. An inactive process is one that is waiting on communication or on a programmed time delay (operations for inactive processes are discussed later in this chapter). An active process is one that is not waiting and is ready to execute.

To manage the active processes, the process scheduler maintains a separate linked list of active processes for each priority level. Instructions are included in the instruction set for starting new processes by adding the process to an active process list. Processes are selected for execution from these active process lists based on the following generalized rules:

- High-priority processes are always executed in preference to low-priority processes. If a low-priority process is executing when a high-priority process is added to the high-priority active process list, the low-priority process is preempted and the high-priority process is executed. A high-priority process is executed until it completes or until it must wait for communication or for a programmed time delay.
- When no high-priority processes are available for execution, a low-priority process may be executed. A low-priority processes is executed until it must wait for communication or for a programmed time delay. In addition, to ensure that one low-priority process does not monopolize the processor, a low-priority process that has been executing for more than about one millisecond is suspended. The suspended process is placed at the end of the low-priority active process list and the process at the beginning of the low-priority active process list is then executed.

2. Floating Point Unit

One version of the Transputer includes hardware for performing floating point arithmetic operations. Internally, the floating point unit includes a three-register floating-point evaluation stack that operates in the same manner as the "normal" or integer processor's evaluation stack. The floating-point unit operates in parallel with the other components of the Transputer. Floating-point operations may, therefore, be performed in the floating-point unit at the same time that integer calculations are being performed in the integer processor. Currently, only the Transputer model T800 includes this floating-point unit.

3. Links

The hardware links provide the means for implementing CSP communication channels between processes executing on different Transputers. A link from one Transputer is connected to a link on another Transputer to provide a bidirectional pair of communications channels. Each hardware link can perform simultaneous, independent input and output communication. Instructions are included in the Transputer's instruction set for performing input and output operations using the links. A block diagram of a communications link is shown in Figure 2.6. Each communications link consists of an independent direct memory access controller and serial communication logic.

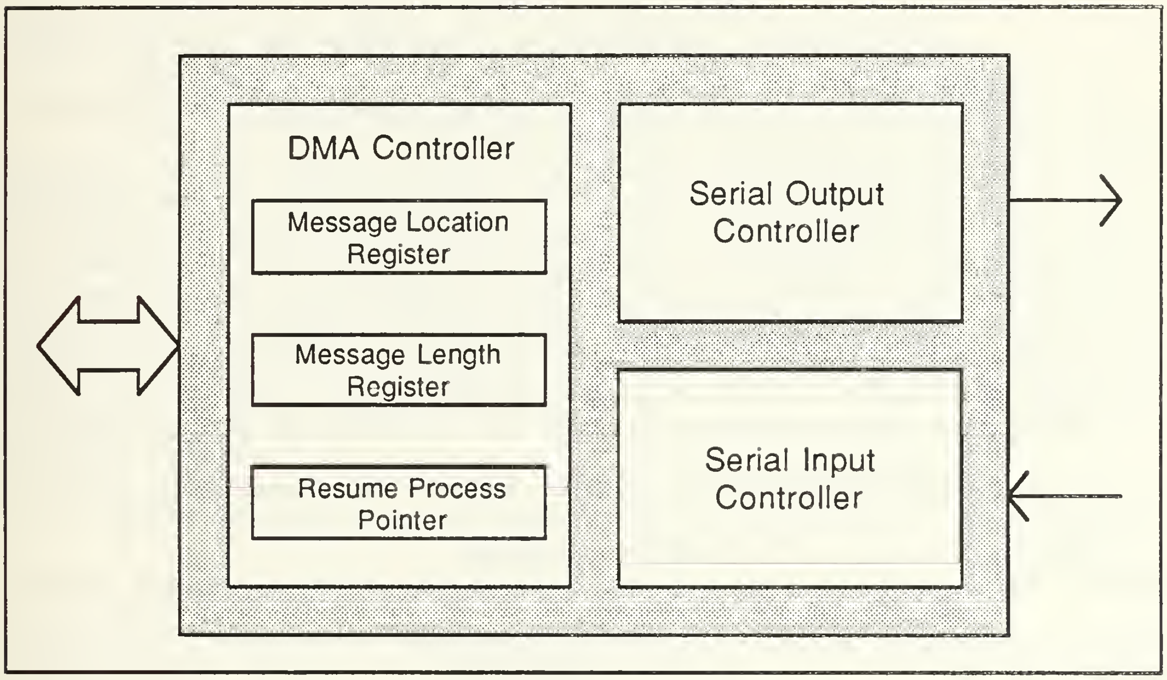


Figure 2.6. **Hardware Communication Link Logical Block Diagram**

To perform a communication via a hardware link, the communication link address and the size and location of a message are specified, then the communications instruction is executed. This initializes the direct memory access controller with the size and location of the message to be communicated. The process executing the communication instruction is suspended and a pointer for later resuming the process is saved (another ready process from an active process list may then be executed). When both the sending and receiving links have been initialized in this manner, the message communication is accomplished. When the communication is complete, the process which was suspended for communication is added to the end of the appropriate high- or low-priority active process list to wait its turn for execution.

Messages are transmitted by the links one byte at a time in a bit-serial format. After a receiver has recognized the reception of a byte and is capable of receiving another byte, the receiver transmits an acknowledge message. The transmitter will await reception of the acknowledge message before transmitting the next message byte. Since the link hardware performs no error checking on messages, the purpose of the acknowledge message is solely to control the flow of message bytes between the links. Figure 2.7 shows a formatted message byte and an acknowledge message.

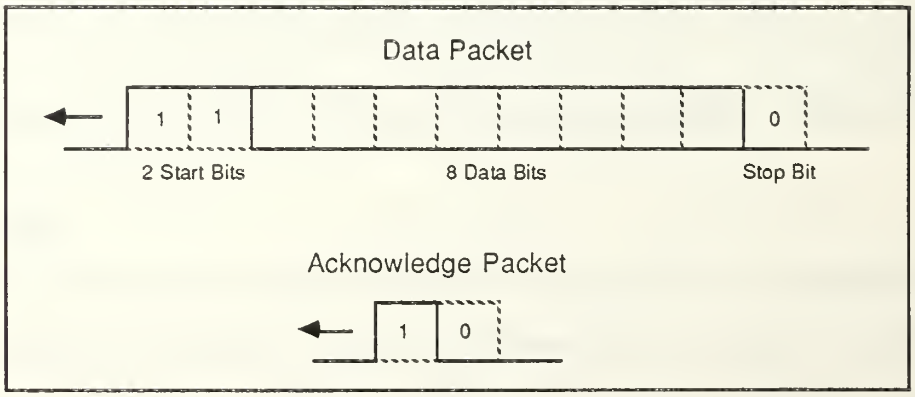


Figure 2.7. **Serial Communication Link Protocol**

A method for communicating between processes executing on the same Transputer is also provided. To perform such a communication, a memory address and the size and location of a message are specified, then the communications instruction is executed (note that this is the same sequence required to initiate communication via a link). When both the sending and receiving processes are ready to communicate, the communication is accomplished by performing a memory-to-memory transfer of the message data.

4. Memory

The Transputer can address four gigabytes of memory. This memory space is divided into two non-overlapping segments: an internal or on-chip segment of memory and an external or off-chip segment of memory. Although these two segments of memory are logically the same, the physical characteristics of the two segments are quite different.

The on-chip memory consists of up to four kilobytes of static random access memory (depending on the type of Transputer being considered). Within the address space, the on-chip memory occupies the lowest block of addresses. The on-chip memory can be accessed for read or write via the internal processor bus in one processor clock cycle.

The off-chip memory forms the balance of the address space. The off-chip memory is accessed via an external memory interface. This external memory interface provides access to the external memory by multiplexing a 32-bit address and 32 bits of data onto a single

32-bit external bus. The timing for this multiplexing slows access to the external bus to a minimum access time of three processor clock cycles. The actual number of cycles required to access external memory is also dependent on the requirements of the external memory devices. For the Transputer systems available in our laboratory, the external memory access times range from three to five processor clock cycles. This difference in access times between the "fast" on-chip and "slow" off-chip memory can affect program performance. Frequently accessed variables or code segments should, therefore, be preferentially located in the "fast" on-chip memory.

Additionally, the external memory interface includes control logic for refreshing external dynamic random access memory devices. Control lines and signals are also provided to facilitate peripheral device direct memory access to the external segment of memory.

5. Timers

The Transputer provides two hardware timers. Each of these timers can be viewed as free-running 32-bit binary counters. One of the timers is accessible to high-priority processes; the other timer is accessible to low-priority processes. The high-priority timer increments at intervals of one µsecond, for a total cycle time of about 72 minutes. The low-priority timer increments at intervals of 64 µseconds, for a total cycle time of about 76 hours.

Instructions are provided for initializing the value of these timers and for reading the current value of a timer. An instruction is also provided for suspending execution of a process until a specified timer value is reached. To implement this instruction, the Transputer maintains a linked list of suspended processes waiting on timer values. Separate lists are maintained for the high- and low-priority timers. These lists are ordered by the specified "wait-until" time value. The first "wait-until" time value in each list is loaded into a dedicated register and, using hardware, is compared with the current value of the high- or low-priority timer. When the "wait-until" timer value is reached, the suspended process is removed from the timer list and is added to the end of the appropriate active process list to wait its turn for execution. The next timer list entry is then loaded for comparison.

6. External Event Input

The external event input on the Transputer is similar to an external interrupt input. To the Transputer, this external event input appears as a communications channel which is capable of transmitting a signal to a user's program. A user's program requesting input from the external event channel will be suspended if the external event input is not being asserted. Then, when the external event input is asserted, the process will be added to the end of an active process list to wait its turn for execution. Either a high- or a low-priority process may request input from the external event channel.

III. TRANSPUTER PERFORMANCE

A. OVERVIEW

As has been described in the previous chapter, the Transputer is a complex microprocessor. Because of this complexity, the performance of even a single Transputer can be affected by many factors. Such factors might include whether or not the program and/or associated data is in on-chip or off-chip memory or if there is internal bus contention resulting from external communications link direct memory access. When considering a network of Transputers, the factors that can affect overall network performance are multiplied considerably.

Developing efficient Transputer-based systems in the face of this complexity requires a firm understanding of Transputer performance and the manner in which different factors influence that performance. Evaluating Transputer-based systems requires accurate methods for measuring the performance of individual and networked Transputers.

To begin to understand Transputer performance characteristics and to gain experience in measuring individual and networked Transputer performance, a series of timing and performance studies was conducted.

B. PRIOR RESEARCH

INMOS provides basic performance specifications for the Transputer [In87b and In87d]. The basic specifications list the performance for individual machine-level and high-level language operations. The listed specifications consider the effects of some of the factors that can potentially affect performance. While it is expected that the manufacturer's performance specifications are accurate, it is necessary to independently confirm this.

Prior theses [Va87 and Ha87] document some detailed tests and analyses of Transputer performance. These theses point out, however, that some aspects of their performance test results do not appear to be consistent or cannot adequately be explained. They suggest that further research be performed in this area to resolve the identified problems and to extend the scope of performance testing.

C. TEST METHODOLOGY

This chapter documents the suggested further timing research and documents the results of an initial set of timing tests on the recently released T800 20 MHz Transputer. Two major categories of testing are addressed. The first category is testing to confirm the basic manufacturer's performance specifications for the Transputer. The second category is testing to determine the interaction that exists between the operation of the central processing unit and communication link direct memory access activity.

1. Configuration

To accomplish both types of testing, a single test configuration was developed. The test configuration consists of a central "target" Transputer and four "satellite" Transputers, each attached to the target Transputer by a communications link. In addition, there are associated Transputers which perform the functions of control and of data routing and recording. A logical diagram of this test configuration is presented in Figure 3.1. A detailed diagram of the test set-up is presented in Appendix A.

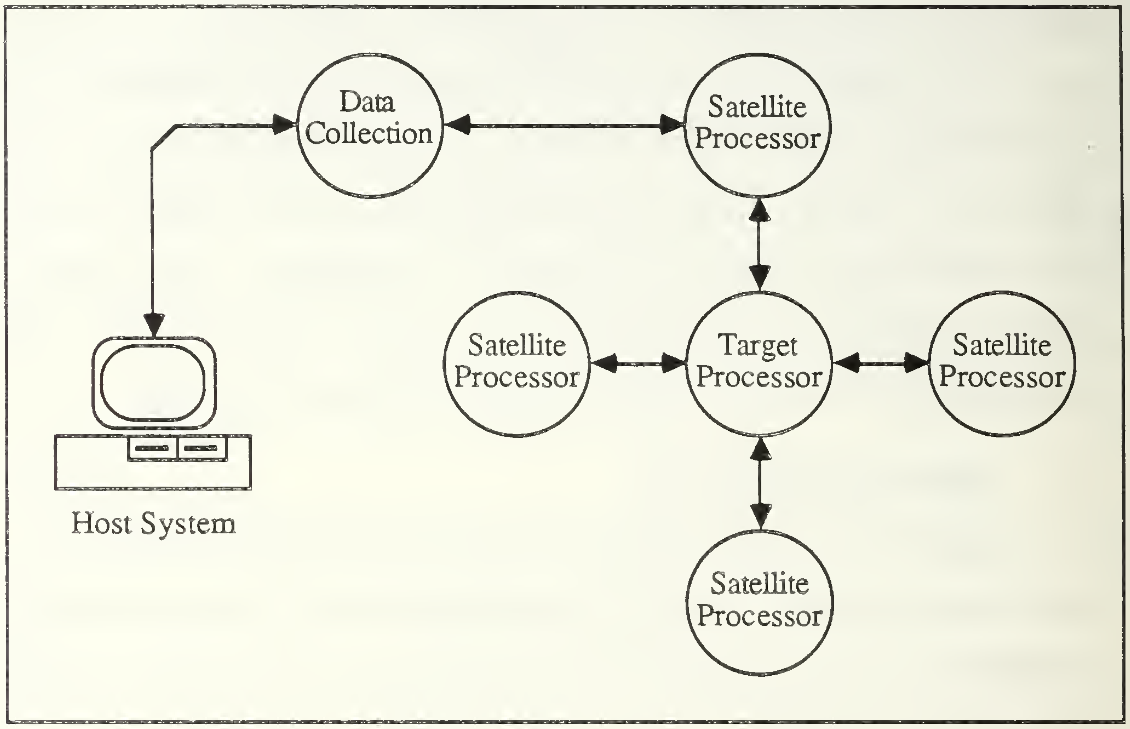


Figure 3.1. **Logical Diagram of Test Configuration**

Although based on and quite similar to a previously used test configuration [Ha87], there is a subtle but significant difference between the two test configurations. In the prior test configuration, one of the satellites was the Transputer installed in the host development system. User programs executed on the Transputer in the host development system are run from within the Transputer Development System (TDS) "shell."

After some initial experimentation, it became apparent that, in some way, this shell affected the timing of programs run from within the shell. The timing of programs running on Transputers external to the shell but depending on communications to or from a program run from within the shell also appeared to be affected. It is postulated that some shell processes are active while the user program is executing and, to an extent, interfere with the user program. References to such processes can be found in [In87a].

Although it may be of interest to research this aspect of TDS in the future, it is sufficient for the purposes of the current timing tests to simply ensure that all timing measurements are performed external to and independent of the host development system Transputer.

2. Software

In general, the target Transputer performed some calculation in a loop while the satellite Transputers placed different link input and output communications loads on the target Transputer. The test software monitored the time required to perform link communications and the number of calculation loops that could be performed during those communications. Appendix A provides a listing of the programs used to measure and record processor performance.

3. Conditions

Because so many factors can interact and affect Transputer performance, it was necessary to set and hold some test conditions fixed so that the effects of variations in parameters of interest could be properly interpreted. Constant for the tests documented by this thesis were:

- T414 Transputer processor speed was 15 MHz.
- T800 Transputer processor speed was 20 MHz.
- Communications link speed was fixed at 20 megabits per second for both Transputer types.
- Program code and scalar variable values in the calculation loop were located in the "fast" on-chip memory for both Transputer types.
- Scalar variables in the calculation loop were in "local" scope (i.e., within the first 16 bytes of the bottom of the workspace and accessible by single byte load and store instructions).
- Memory-to-memory transfer blocks were located in "slow" off-chip memory for both Transputer types.
- Communications data blocks were fixed at 100,000 bytes. Because this block size is much greater than the size of the on-chip memory, the block was located in "slow" off-chip memory for both Transputer types.
- Communications processes were run at high priority and the calculation loop was run at low priority.
- Time measurements were taken in the each Transputer using the high-priority one-μsecond resolution timer.

The effects of varying certain parameters of interest were investigated. The following list identifies the parameters of interest and the manner in which they were varied.

- The characteristics of the target Transputer calculation loop were varied by placing different types and numbers of operations within the loop. The operations used within the loop were no operation, assignment, addition, subtraction, multiplication, division, and 100- and 1000-byte memory-to-memory transfers. The number of individual operations of a type in a loop ranged from 0 to 4 operations per loop.

- Communications conditions were varied by changing the number of links that were active at one time. Additionally, the communications conditions were varied by changing the size of a communications packet. The number of active communications links ranged from 0 to 4 input links active and from 0 to 4 output links active. The individual data packet size was varied in 16 steps from one byte per packet to 100,000 bytes per packet with an overall constant data block size of 100,000 bytes.

4. Data Management

From the number of possible combinations of test conditions, it was clear quite early that these timing tests would generate a large amount of test data. To more effectively handle this test data, it was decided to record the data in some database management system. Using a database management system provided for ease of access to selected aspects of the data. Additionally, since the potential exists for performing further timing tests, such a system will facilitate the incorporation and aggregation of any future timing data.

Since the Ingres relational database management system was readily available on the departmental mini-computer, this system was selected for use. To facilitate loading of the timing database, conditions for a particular timing test and the test results were directed to a disk file on the host development system, then electronically transferred to the departmental mini-computer for loading into the Ingres timing database. Appendix B describes the Ingres database created for the timing data and gives some examples of accessing timing information from the database.

D. TEST RESULTS

Test runs covering the range of conditions described previously were completed and the resulting data was loaded into the Ingres timing test database. Data was extracted from this database to "view" several aspects of Transputer performance. These aspects of performance are:

1. Isolated Processor Performance

The first view of the timing results is to compare measured instruction execution times with the manufacturer's specified instruction times for a selected subset of Transputer instructions. This is intended as a confirmation of the manufacturer's stated execution times. Additionally, if the timing results are consistent with the manufacturer's specifications, it will tend to validate the timing test methodology that is being used.

Determining the manufacturer's specified execution time for a selected loop operation requires that the operation be examined at the machine-language level. Using a disassembler [Br87] to facilitate examination of the timing program object code, each of the selected operations was decomposed into its machine-level components. The manufacturer-specified number of processor clock cycles for each of these components of an operation was summed, then multiplied by the period of the processor clock. The result of this calculation is the expected execution time for a particular loop operation executed in on-chip memory. For example, the expected execution time for the multiply operation is determined as follows:

Multiply Operation:

```
    a := b * c
```

Machine-Language Equivalent Multiply Operation:

```
    ldl          b        (2  cycles)
    ldl          c        (2  cycles)
    pfix; mult            (1 + 38 cycles)
    stl          a        (1  cycle)
```

Execution Time Calculation (T800 @ 20 MHz):

$$44 \text{ cycles} \times \frac{0.050\ \mu\text{sec}}{\text{cycle}} = 2.200\ \mu\text{sec}$$

Note that execution of instructions in "slow" off-chip memory affects the calculation of expected execution times. To determine the expected time for such an off-chip operation, the same basic method described above is used, except that a separate accounting of on-chip and off-chip cycle counts is maintained. The off-chip cycle count is then multiplied by a hardware-dependent scale factor. This scale factor is the number of processor cycles required to make a single access to the off-chip memory. In the case of the hardware used for these timing tests, this scale factor is 4 for the T414 Transputer and 5 for the T800 Transputer.

In addition, in the particular hardware configuration used for these timing tests, off-chip memory consisted of dynamic random access memory devices. Such devices must periodically be "refreshed" to maintain their data. Although this refresh operation is relatively fast and is handled automatically by the Transputer hardware, access to the off-chip memory is restricted during a short period of time. In the worst case, this increases the average time required to access off-chip memory by a factor of 1.0237 for the T414 Transputer and 1.0213 for the T800 Transputer [In87b]. The worst-case overall time for an off-chip operation is, then, the sum of the on-chip time and the scaled and "refresh delayed" off-chip time. An example calculation for execution of an off-chip memory-to-memory transfer follows:

Memory-to-Memory Transfer Operation:

```
[array FROM 0 FOR 1000] := [array FROM 0 FOR 1000]
```

Machine Language Equivalent Operation:

```
pfix; pfix; ldc    #$800    (1 + 1 + 1 cycles)
pfix; mint                  (1 + 1 cycles)
wsub                        (2 cycles)
stl                temp     (1 cycle)
pfix; pfix; ldc    #$800    (1 + 1 + 1 cycles)
pfix; mint                  (1 + 1 cycles)
wsub                        (2 cycles)
ldl                temp     (2 cycles)
pfix; pfix; ldc    #$3E8    (1 + 1 + 1 cycles)
pfix; move                  (1 + 8 cycles maximum
                            500 off-chip cycles)
```

Execution Time Calculation (T414 15 MHz):

$$29 \text{ cycles x } \frac{0.067\ \mu\text{sec}}{\text{cycle}} = 1.933\ \mu\text{sec}$$

$$500 \text{ cycles} \times \frac{0.067\mu\text{sec}}{\text{cycle}} \times (4 \times 1.0237) \text{ off-chip} = 136.493\ \mu\text{sec}$$

$$1.933\ \mu\text{sec} + 136.493\ \mu\text{sec} = 138.426\ \mu\text{sec maximum}$$

Measured execution times for different operations were derived from the timing data as follows:

- Consider only timing data where no external communication links were active.
- Calculate the average time per loop for a test set with no operations in the loop. (The average loop time is simply the time of measurement divided by the number of loops executed during that time.)
- Extract the average time per loop for a test set with 1, 2, 3, and 4 instances of a selected operation in the loop.
- Subtract the no-operation loop time from each of the selected operation loop times. These loop times have now been "adjusted" to remove any loop overhead time.
- The incremental increase in adjusted loop times for the selected operation loops is the expected time required to perform a single operation.

Tables 3.1 and 3.2 list the expected and measured execution times for the selected loop operations. These results show that, except for the divide operation, the measured results are consistent with the expected results. The expected time for the divide operation is based on the operation taking 39 processor clock cycles [In87d]. The measured results indicate that the divide operation takes 38 processor clock cycles. Subsequent to performing the timing tests, it was confirmed with the manufacturer that the divide operation, in fact, takes 38 clock cycles as measured in the timing tests [Pe88].

TABLE 3.1

T414 EXPECTED AND MEASURED INSTRUCTION EXECUTION TIMES

Operation	Operations per Loop	Test Duration (μsec)	Loops During Test	Loop Time (μsec/Loop)	Measured Op Time (μsec)	Calculated Op Time (μsec)
Null Loop	0	1000014	214153	4.67		
Assignment	1	1000013	205352	4.87	0.20	0.200
	2	1000014	197246	5.07	0.20	
	3	1000011	189755	5.27	0.20	
	4	1000012	182813	5.47	0.20	
Addition	1	1000013	197246	5.07	0.40	0.400
	2	1000012	182813	5.47	0.40	
	3	1000014	170349	5.87	0.40	
	4	1000011	159475	6.27	0.40	
Subtraction	1	1000013	197246	5.07	0.40	0.400
	2	1000012	182813	5.47	0.40	
	3	1000015	170349	5.87	0.40	
	4	1000011	159475	6.27	0.40	
Multiplication	1	1000011	131497	7.61	2.94	2.933
	2	1000016	94878	10.54	2.94	
	3	1000022	74212	13.48	2.94	
	4	1000017	60938	16.41	2.94	
Division	1	1000013	129230	7.74	3.07	3.133
	2	1000012	92535	10.81	3.07	
	3	1000018	72071	13.88	3.07	
	4	1000022	59019	16.94	3.07	
Block Move (100 Bytes)	1	1000030	50711	19.72	15.05	15.516 (maximum)
	2	1000036	28773	34.76	15.04	
	3	1000042	20077	49.81	15.05	
	4	1000074	15433	64.80	15.03	
Block Move (1000 Bytes)	1	1000117	7007	142.73	138.06	138.426 (maximum)
	2	1000100	3562	280.77	138.05	
	3	1000056	2388	418.78	138.04	
	4	1000066	1796	556.83	138.04	

TABLE 3.2

T800 EXPECTED AND MEASURED INSTRUCTION EXECUTION TIMES

Operation	Operations per Loop	Test Duration (µsec)	Loops During Test	Loop Time (µsec/Loop)	Measured Op Time (µsec)	Calculated Op Time (µsec)
Null Loop	0	1000009	285581	3.50		
Assignment	1	1000010	273845	3.65	0.15	0.150
	2	1000008	263035	3.80	0.15	
	3	1000009	253038	3.95	0.15	
	4	1000010	243789	4.10	0.15	
Addition	1	1000008	263035	3.80	0.30	0.300
	2	1000011	243789	4.10	0.30	
	3	1000010	227167	4.40	0.30	
	4	1000011	212667	4.70	0.30	
Subtraction	1	1000009	263035	3.80	0.30	0.300
	2	1000010	243789	4.10	0.30	
	3	1000011	227167	4.40	0.30	
	4	1000010	212667	4.70	0.30	
Multiplication	1	1000011	175357	5.70	2.20	2.200
	2	1000015	126524	7.90	2.20	
	3	1000012	98964	10.11	2.20	
	4	1000010	81263	12.31	2.20	
Division	1	1000013	172334	5.80	2.30	2.350
	2	1000015	123400	8.10	2.30	
	3	1000009	96109	10.41	2.30	
	4	1000016	78704	12.71	2.30	
Block Move (100 Bytes)	1	1000013	57789	17.31	13.80	14.166 (maximum)
	2	1000009	32118	31.14	13.82	
	3	1000044	22262	44.92	13.81	
	4	1000036	17037	58.70	13.80	
Block Move (1000 Bytes)	1	1000012	7558	132.31	128.81	129.110 (maximum)
	2	1000099	3830	261.12	128.81	
	3	1000100	2565	389.90	128.80	
	4	1000524	1930	518.41	128.73	

2. Isolated Communications Link Performance

The Transputer manufacturer also provides specifications for communications link performance [In87b and In87c]. The measured performance of a communications link can also be calculated from timing test data. Measured communications link performance is calculated by dividing the size of the largest single data block communicated over a single link by the time required for the communication. The largest block size was selected to minimize the affects of any communications set-up overhead that might exist. For the timing tests performed, the largest block size was 100,000 bytes. The specified and measured communications data rates are compared in Table 3.3. This table shows that the specified and measured data rates are consistent.

TABLE 3.3

ISOLATED COMMUNICATION LINK PERFORMANCE

Processor Type	Communication Mode	Specified Rate (Kbytes/sec)	Measured Rate (Kbytes/sec)
T414	Input	800	759
	Output	800	762
	Input/Output	1600	1505
T800	Input	1740	1738
	Output	1740	1737
	Input/Output	2350	2344

Of particular interest here is the bidirectional data rate for the T800 Transputer. For the T414 Transputer, the bidirectional data rate is approximately 2 times the unidirectional data rate. For the T800, however, the bidirectional data rate is only about 1.35 times the unidirectional data rate. Since in the worst case of the external off-chip memory the processor bus data rate is at least 20 megabytes per second, accessing data from memory should not limit link performance. Taking a closer look at the link communications protocol provides the key to understanding this problem.

In unidirectional communication for the T800, the acknowledge packet from the receiving Transputer is returned to the transmitting Transputer before the transmitter completes its transmission. Because of this, the transmitter is able to transmit bytes "head-to-tail" without any intervening delays. This results in a maximum theoretical unidirectional data rate of

$$\frac{20 \text{ Mbits}}{\text{sec}} \times \frac{1 \text{ byte}}{11 \text{ bits transmitted}} = \frac{1.82 \text{ Mbytes}}{\text{sec}}$$

which is consistent with the actual unidirectional link data rate. In bidirectional communications, however, each Transputer must "sandwich" acknowledge packets between data packets. This increases the total number of bits transmitted by a Transputer per data byte from 11 to 13 and results in a maximum theoretical bidirectional data rate of

$$\frac{20 \text{ Mbits}}{\text{sec}} \times \frac{1 \text{ byte}}{13 \text{ bits transmitted}} \times 2 = \frac{3.08 \text{ Mbytes}}{\text{sec}}$$

This is less than double the unidirectional rate. Additionally, because a receiving Transputer is also transmitting its own data, it may not immediately be able to return an acknowledge packet for data received. The transmitter may, therefore, be delayed in transmitting its next data byte, further reducing the data rate. However, since the internal timing threshold requirements of the link hardware are not known, it is not possible to exactly quantify this effect.

Because of this bidirectional link communications limitation, two separate unidirectional communications links between T800 Transputers will provide about a 1.1 megabyte greater bidirectional communications throughput than will a single communications link operated bidirectionally.

Since the T414 unidirectional communication rate is significantly less than the maximum possible rate for a 20 Mbit/second data link, space already exists between successive transmitted packets to "insert" an acknowledge packet for a received data packet. Because of this, the T414 does not show the prominent bidirectional communications protocol limitation shown for the T800.

3. Communication Link Interaction

Prior research [Va87] has shown that, for the T414 Transputer, there is minimal interaction between links when multiple links are operated simultaneously. It was desired to determine whether this is also a characteristic of the T800 Transputer with its greater communication link data rates. Figures 3.2 and 3.3 show communications link performance for both the T414 and the T800 under conditions when multiple links are active. Except for the bidirectional protocol limitation previously discussed, these figures show that there is minimal interaction when multiple links are being operated. Note that in Figure 3.2, the number of links active includes both unidirectionally and bidirectionally active links. Because of this, each number of active links may include more than one data point. For example, two links active on Figure 3.2 include the data points for both the case when two unidirectional links are active and the case when one bidirectional link is active.

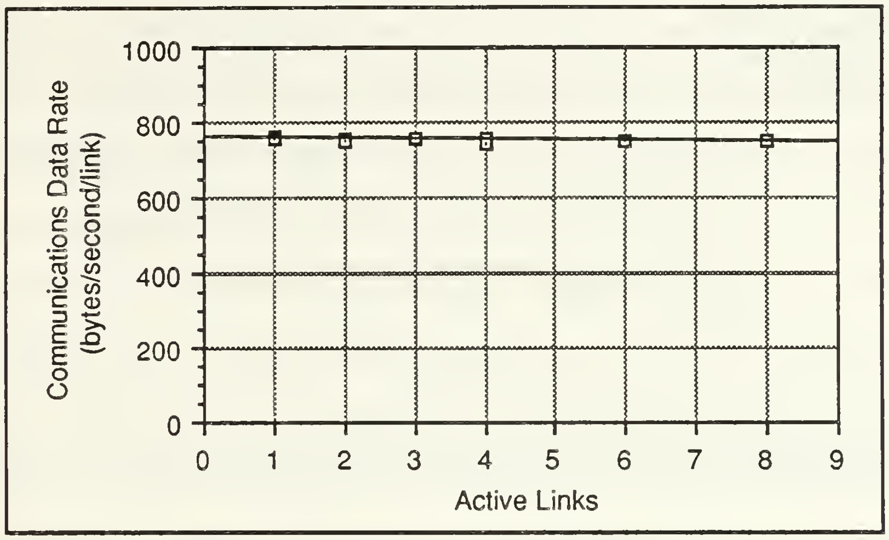


Figure 3.2. **T414 Multiple Link Effects on Communications Rate**

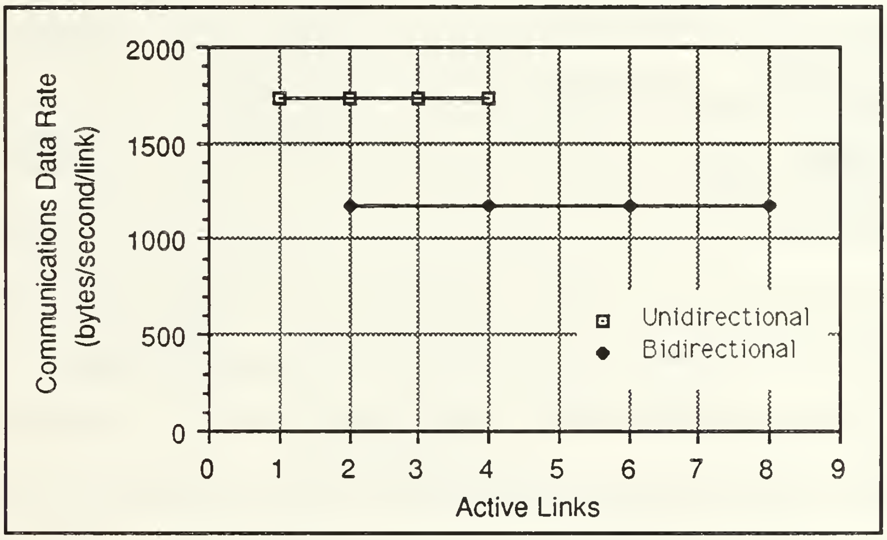


Figure 3.3. **T800 Multiple Link Effects on Communications Rate**

Additionally, for the T414, it has been shown that individual data packet size can affect the overall communications data rate [VA87]. In general, as the data packet size is decreased, the overall communications data rate also decreases. This is because the overhead time associated with the set-up of a data transmission is more significant when the size of the data packet to be communicated is small. The current test configuration and software was used to extend the scope of testing to further investigate this phenomenon. The results of this testing for the T414 and the T800 are shown in Figures 3.4 and 3.5. These graphs, as well as others in this chapter, were conducted with various numbers of links active. The number of links active for a particular graph are identified on that graph. Figures 3.4 and 3.5 show that communications throughput decreases rapidly when the packet size decreases below a threshold point of from about 200 down to 50 bytes per packet.

Further, note the communications characteristics shown in Figure 3.5. This graph shows the protocol limited nature of bidirectional link operations for the T800 Transputer. Compare the plots for the two bidirectional links case with those of the four unidirectional links case. In either of these cases, a total of four links are operating. As packet size increases from one byte per packet, the plots for both links are at first identical. However, as the packet size passes about 20 bytes per packet, the two plots begin to diverge. The four unidirectional link case data rate continues to increase but the bidirectional link case becomes protocol limited and data rate levels off.

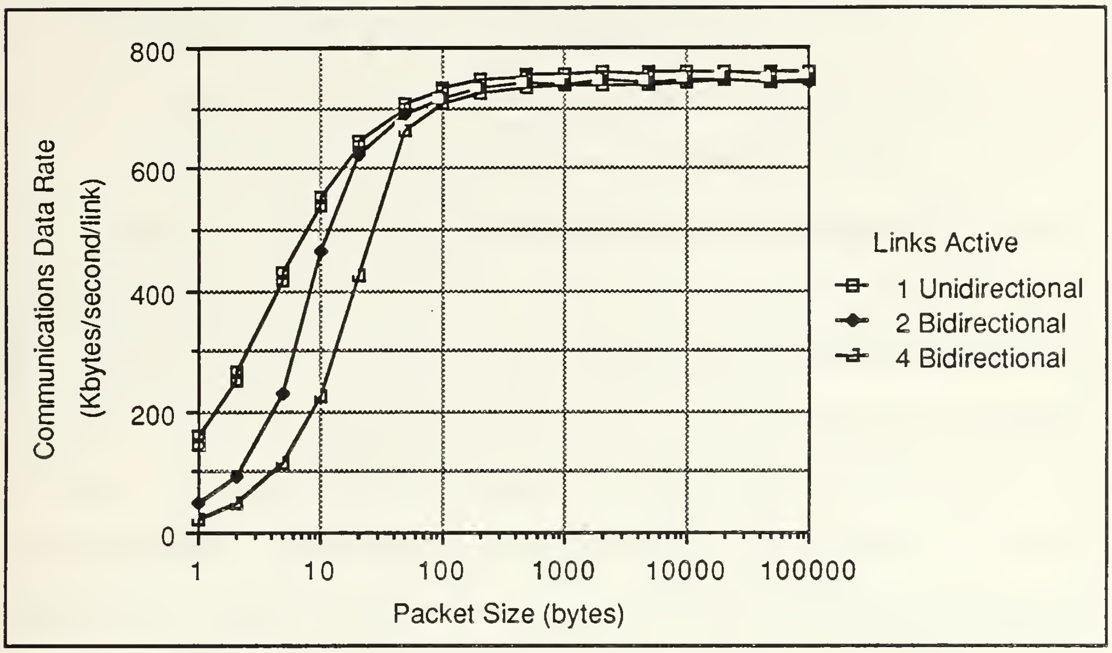


Figure 3.4. **T414 Packet Size Effects on Communication Rate**

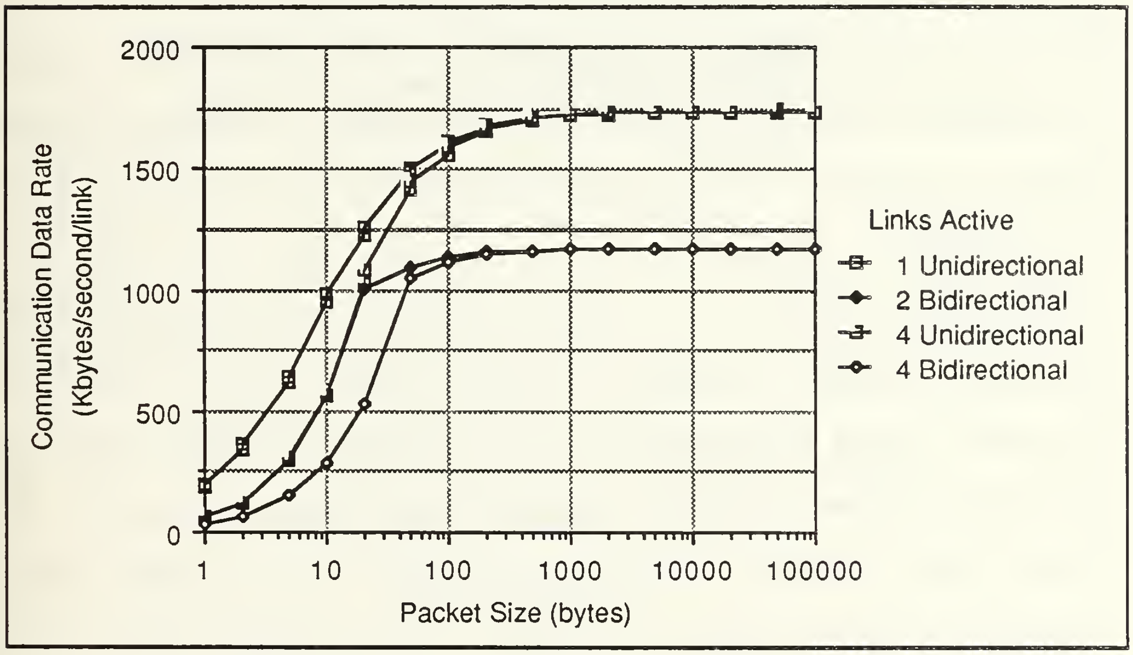


Figure 3.5. **T800 Packet Size Effects on Communication Rate**

4. Communication Link Effects on Processor Performance

One of the key questions raised about Transputer performance has been the extent to which communications link activity interferes with processor execution speed. There are two ways in which such interference can occur. First, both the processor and the communication links contend for access to the same internal data bus. Secondly, the set-up of a communication and the rescheduling of a process upon completion of its communication require some processor overhead. Recalling the timing test methodology, where operations in a loop were conducted in parallel with a variety communications load conditions, timing data is available to address this issue.

For the purposes of comparing processor performance, the average number of loops that were performed per unit time under each set of test conditions was taken as the relative measure of target processor performance. For each set of test conditions, processor performance has been normalized by dividing performance measurements by the "no communication" performance for that set of test conditions.

Figures 3.6, 3.7, 3.8 and 3.9 present representative results from this analysis. As can be seen from these figures, processor performance drops dramatically as the communications packet size is decreased below a certain threshold value. In some cases, performance was reduced to the point where the processor made no progress on its looping calculation; the processor was completely communications bound.

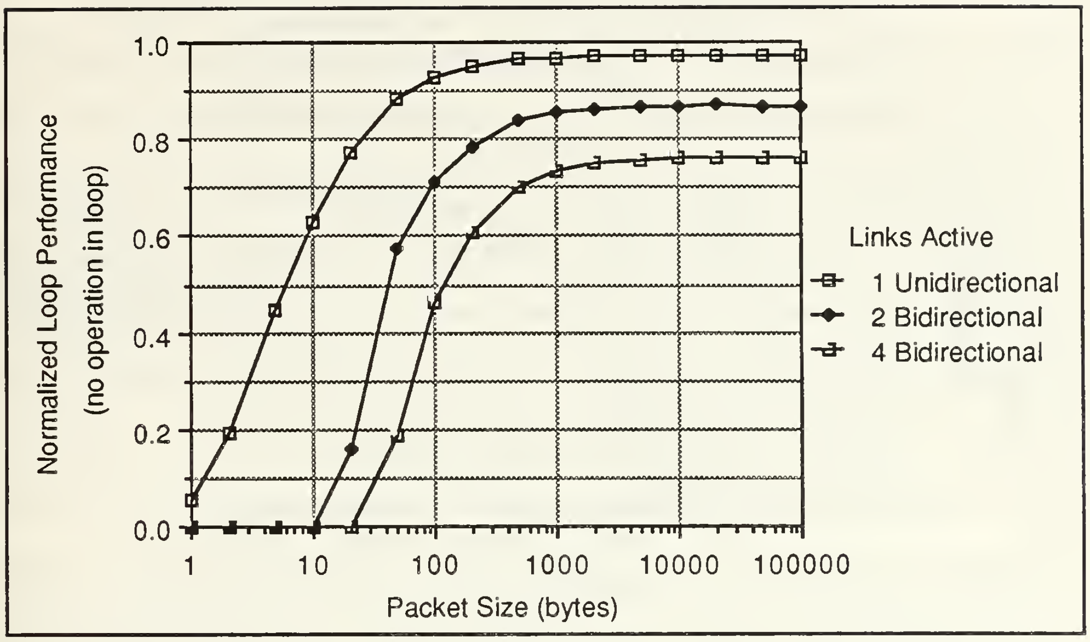


Figure 3.6. **T414 Performance Degradation with No Loop Operation**

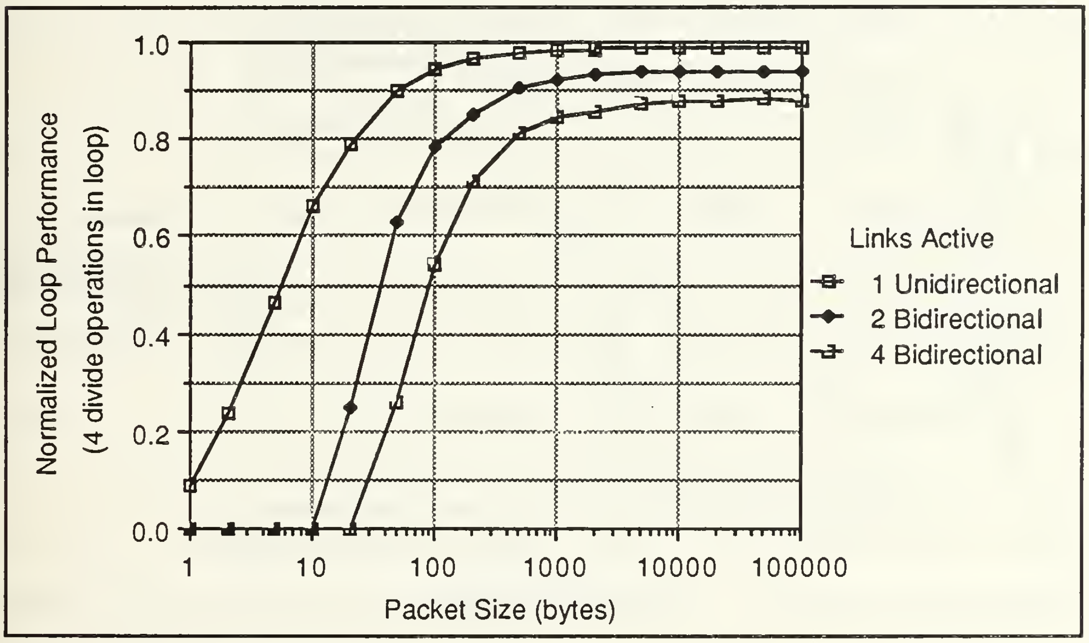


Figure 3.7. **T414 Performance Degradation with Four Divide Operations**

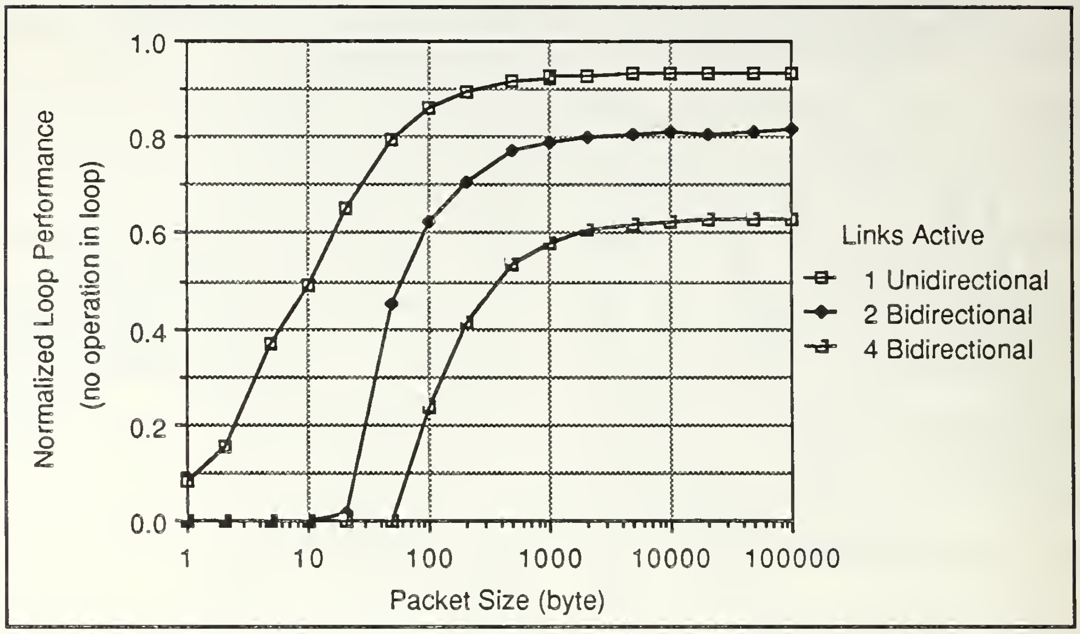


Figure 3.8. **T800 Performance Degradation with No Loop Operation**

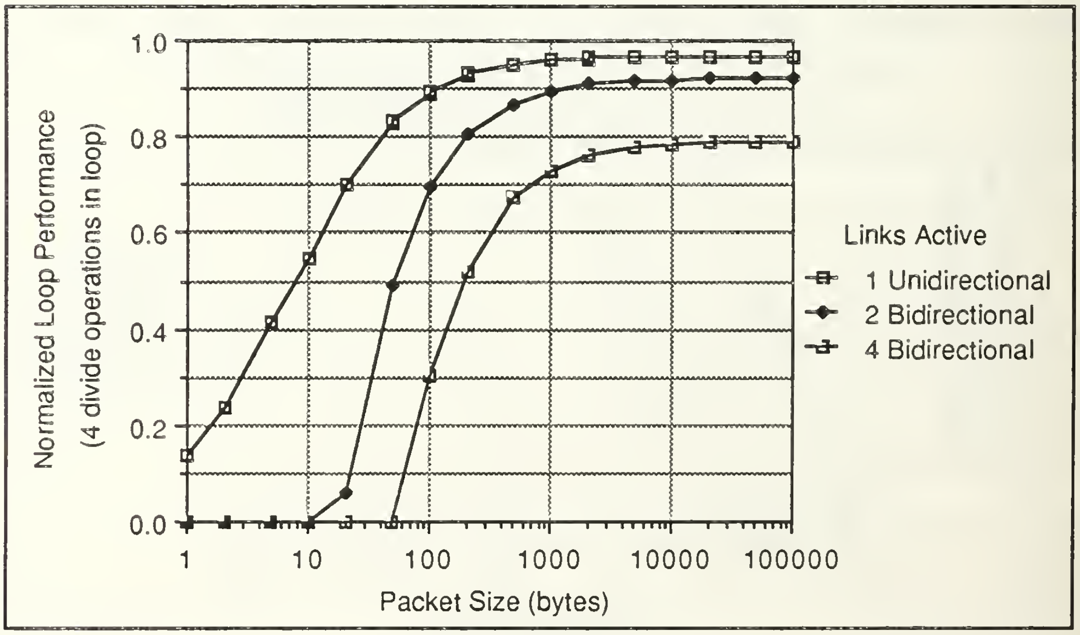


Figure 3.9. **T800 Performance Degradation with Four Divide Operations**

In the left-hand region of the performance graphs, where the individual data packet size is small, the processor overhead associated with the communication is the primary contributor to performance degradation. As the data packet size is increased, the time associated with actually transmitting the data packet increases. As the actual packet transmission time increases, the overhead associated with communicating a packet becomes less significant. Therefore, when the packet size is very large, any processor performance degradation will be due to processor and communications link internal bus contention. Towards the right-hand side of the performance graphs, processor performance degradation can be seen asymptotically approaching this bus contention-only degradation characteristic.

As can be seen by comparing the different performance graphs, the type of calculation performed within the loop also affects the amount of performance degradation. Although there is only a minimal difference in the overhead limited (left-hand) area of the graph, there is a noticeable difference in the bus contention limited (right-hand) portion of the graph. In general, the difference in the bus contention limited area of the graph is directly related to the frequency of memory access required by the calculation loop. For short instructions (such as assignment) that require frequent bus access, the performance degradation is greater than for long instructions (such as division) that require infrequent bus access. Figures 3.10 and 3.11 illustrate this characteristic.

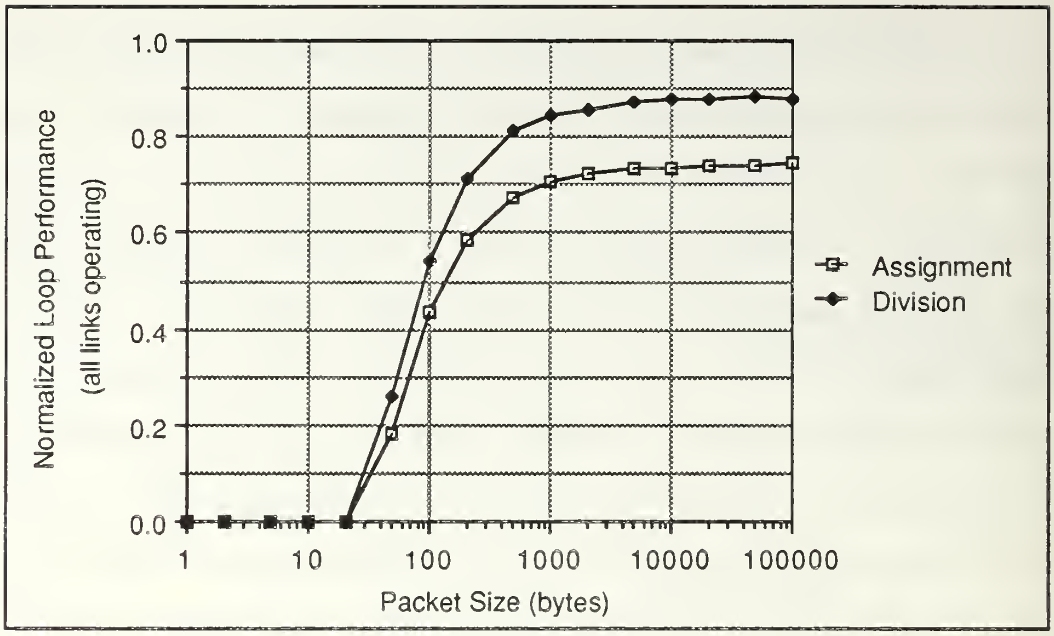


Figure 3.10. **T414 Bus Access Frequency Effects on Performance Degradation**

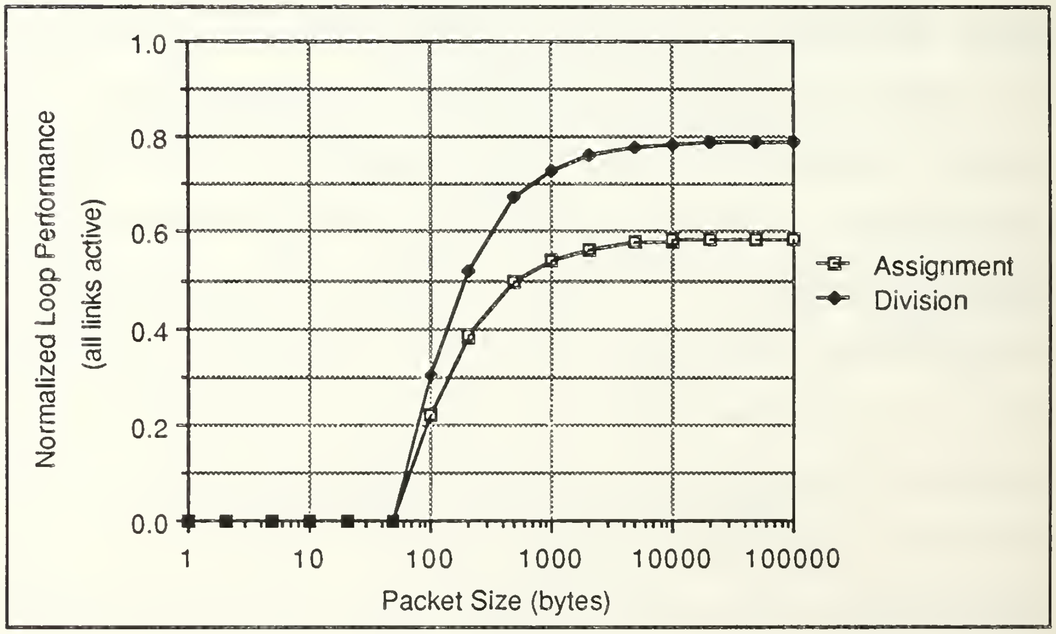


Figure 3.11. **T800 Bus Access Frequency Effects on Performance Degradation**

5. Processor Effects on Communications Link Performance

As a corollary to the question of communications effects on processor performance, it is reasonable to ask what effect an actively executing process has on the performance of a communications link. It might be expected that if the processor is slowed by the effects of communication link bus contention, a communication link would also be slowed by the effects of processor bus contention. Data was extracted from the timing database to determine if such effects did in fact exist. A typical sample of this data is shown in Table 3.4.

TABLE 3.4

PROCESSOR EFFECTS ON COMMUNICATIONS PERFORMANCE

	Data Packet Size (bytes/packet)	No Calculation Data Rate (bytes/second/link)	With Calculation Data Rate (bytes/second/link)
T414	1	23.561	23.561
	10	226.078	226.072
	100	705.184	701.262
	1000	738.460	738.165
	10000	742.242	742.975
	100000	747.049	740.988
T800	1	31.415	31.415
	10	282.415	282.416
	100	1120.561	1112.013
	1000	1166.943	1165.773
	10000	1171.413	1171.097
	100000	1171.866	1171.646

Table 3.4 shows that the effects of processor activity on communication link performance is not significant. The reason for this is that, when the communicating process is of an equal or higher

priority to the executing process, the communication link has priority access to the data bus [In86e]. The communication link is, therefore, virtually unaffected by bus contention. Bus arbitration priority was defined in this manner so that the lower bandwidth communications links would not be idled [Ha87].

6. Summary

Through the timing tests that have been performed, it has been seen that a strong relationship exists between the calculational performance of the Transputer and the operation of its communications links. Additionally, the size of individual communications packets has a pronounced effect on both the calculational and communications performance of the Transputer. Knowledge of these characteristics can be used by the software designer to develop more efficient software.

IV. SHARED MEMORY MODEL PROGRAMMING INTERFACE

A. BACKGROUND

The concept of Communicating Sequential Processes as implemented by the Transputer is one methodology for interprocess communication and synchronization. An alternative concept utilizes globally shared memory as a means for interprocess communication and synchronization. A body of software and development experience that makes use of this shared memory concept exists and could be useful in developing programs for a network of Transputers. In particular, distributed programs developed in the ADA language environment use a shared memory model.

In a large network of Transputers, the physical connectivity of the network is limited by the availability of only four bidirectional communications links per Transputer. This limited connectivity often imposes restrictions upon the distributed systems software designer. The software designer must be aware of and consider the physical configuration of the network. As a result, the designer must often work around the limitations that the configuration imposes. The use of a shared memory model in a network of Transputers is one methodology that might be used to isolate the software designer from these physical configuration limitations.

These factors have motivated the development of prototype software to implement a shared memory model environment in a network of Transputers. The purpose of this prototyping effort was to gain experience in developing such a system for a distributed system. This chapter describes the design, implementation, and evaluation of this prototype.

B. EVENT COUNTS AND SEQUENCERS

When multiple processes are sharing a common resource, as in the case of a shared memory system, some means must be available for the processes to coordinate or synchronize their use of the resource. One means for this synchronization is through the use of Event Counts and Sequencers [ReKa79].

As its name implies, an event count is a value representing the cumulative total number of events of a particular type that have occurred in a system. An event count might be used, for example, to record and monitor the number of times a particular shared memory location has been written to or modified. Many separate event counts may be defined in a system for use in monitoring different types of events.

A sequencer is also a value representing a count. This count is used to control the sequence in which events occur. Each sequencer count value can be thought of as a reservation for a process to use a shared resource. These reservations are issued in sequencer count order to requesting processes. Processes with reservations are then permitted access to the controlled resource in sequencer count order. For example, several processes writing to the same memory location may use a sequencer to ensure that only one process writes to the location at a time. As with event counts, many sequencers may be defined in a system to control access to different shared resources.

Several primitive operations are defined for event counts and sequencers. These operations are:

- read(event_count)—The read operation returns the current value of a specified event count.
- advance(event_count)—The advance operation increments the value of a specified event count.
- await(event_count, count_value)—The await operation suspends the process executing the await command until the value of a specified event count has at least reached the identified count value. Note that execution of the suspended process is not necessarily resumed immediately when the specified count value is reached. The process might not resume until some time after the count is reached.
- ticket(sequencer)—The ticket operation returns the value of the next available reservation "slot" for a specified sequencer.

C. IMPLEMENTATION

The model of event counts and sequencers was selected as the paradigm for implementing the prototype shared-memory model. Since a network of Transputers does not physically share any memory, the implementation must make use of software to simulate a sharing of memory. The core of the implementation is a distributed software kernel which executes on each node in the network. Figure 4.1 depicts the general physical configuration of a network with this kernel.

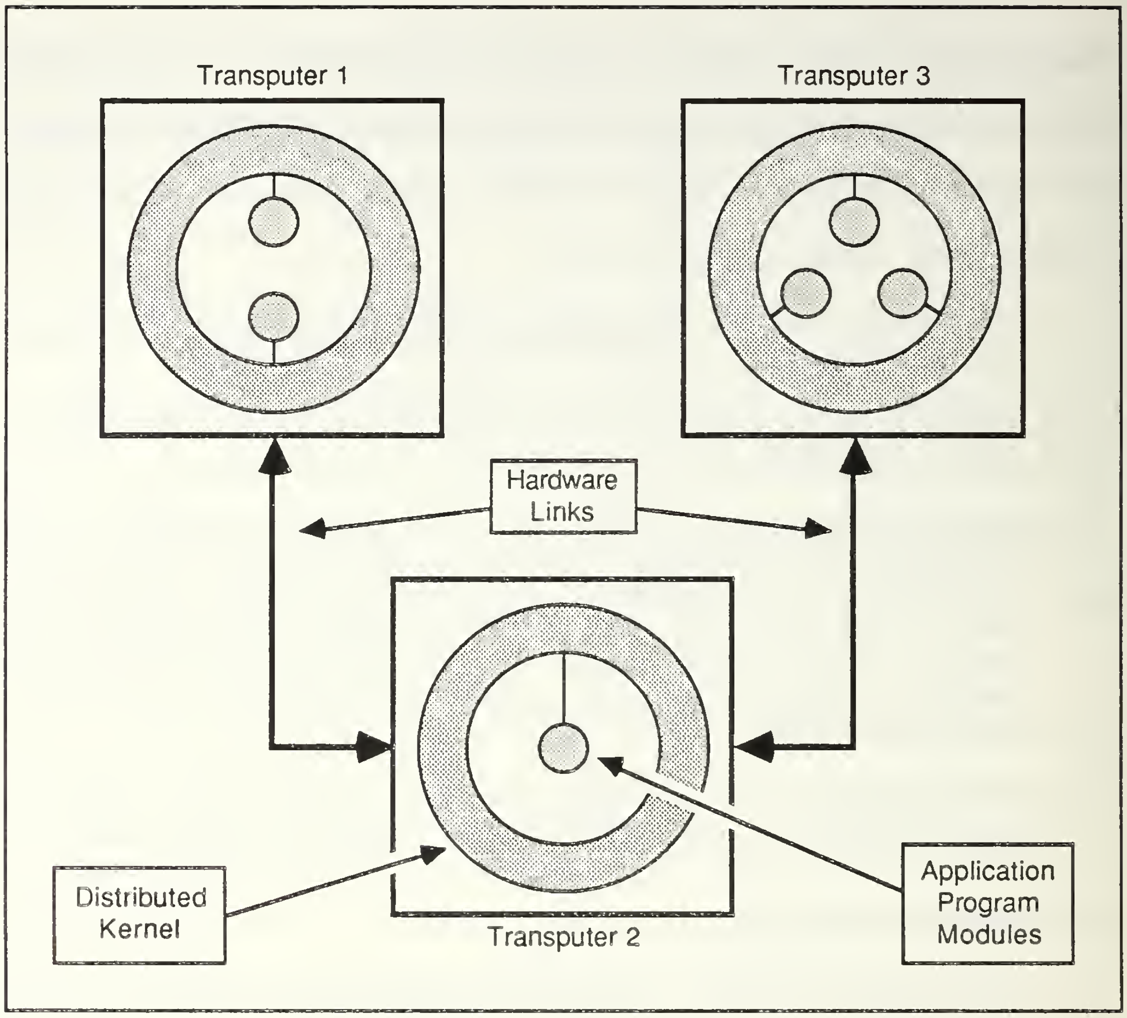


Figure 4.1. **Physical Configuration of Shared Memory Kernel**

The kernel contains the system's shared memory and maintains the system's event count and sequencer values. The system's shared memory as well as the event count and sequencer values are apportioned amongst the kernels operating on different nodes. The kernel also "hides" the specific physical configuration of the network from the application program by managing all communications between program modules and network nodes. Figure 4.2 shows the distributed application program's view of the kernel. A detailed description of the kernel is provided in this section.

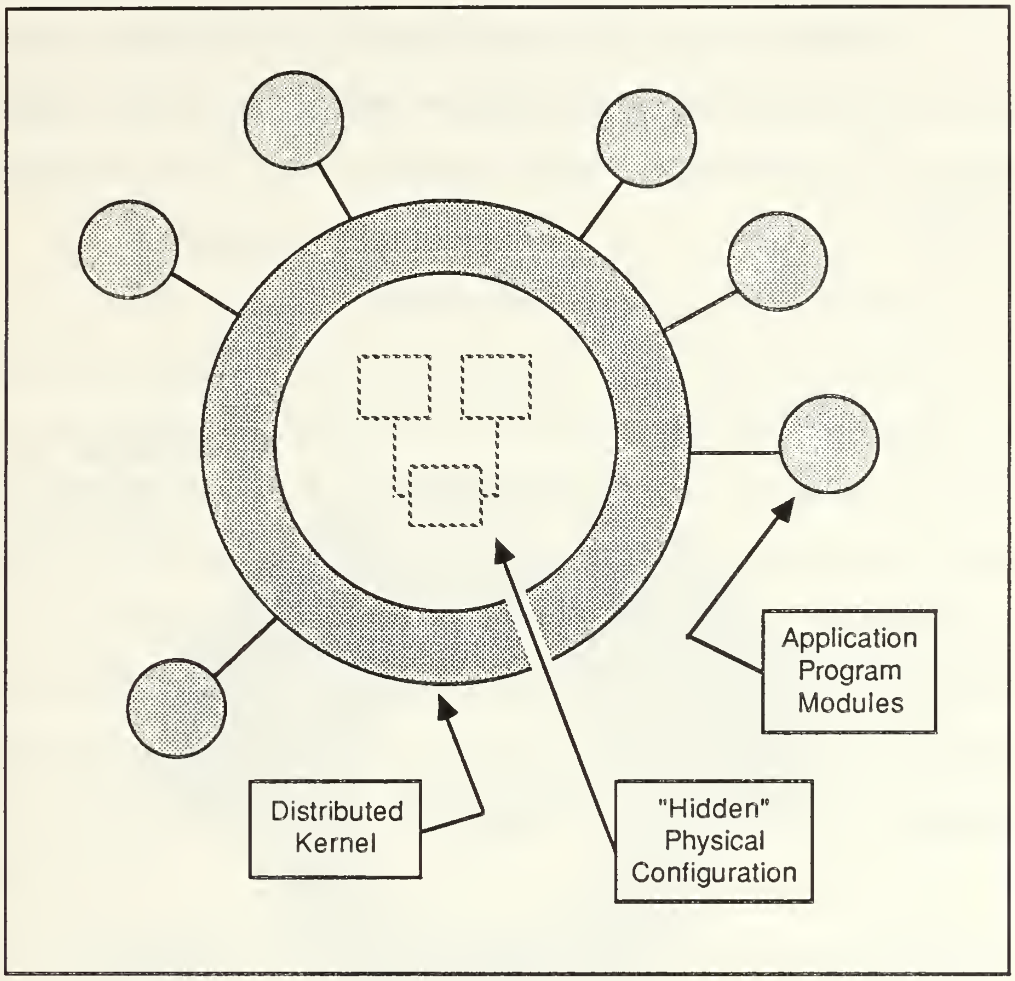


Figure 4.2. **Logical Configuration of Shared Memory Kernel**

The prototype implementation also includes a library of procedures for interfacing with the kernel. This library includes the basic primitive operations defined for event counts and sequencers. In addition, primitive operations have been defined and included for accessing shared memory segments. The procedures in this library are also listed in this section.

1. The Kernel

Figure 4.3 shows a logical diagram of the software kernel. As shown, the kernel consists of three major parts: the input/output buffers, the communications manager, and the shared memory manager.

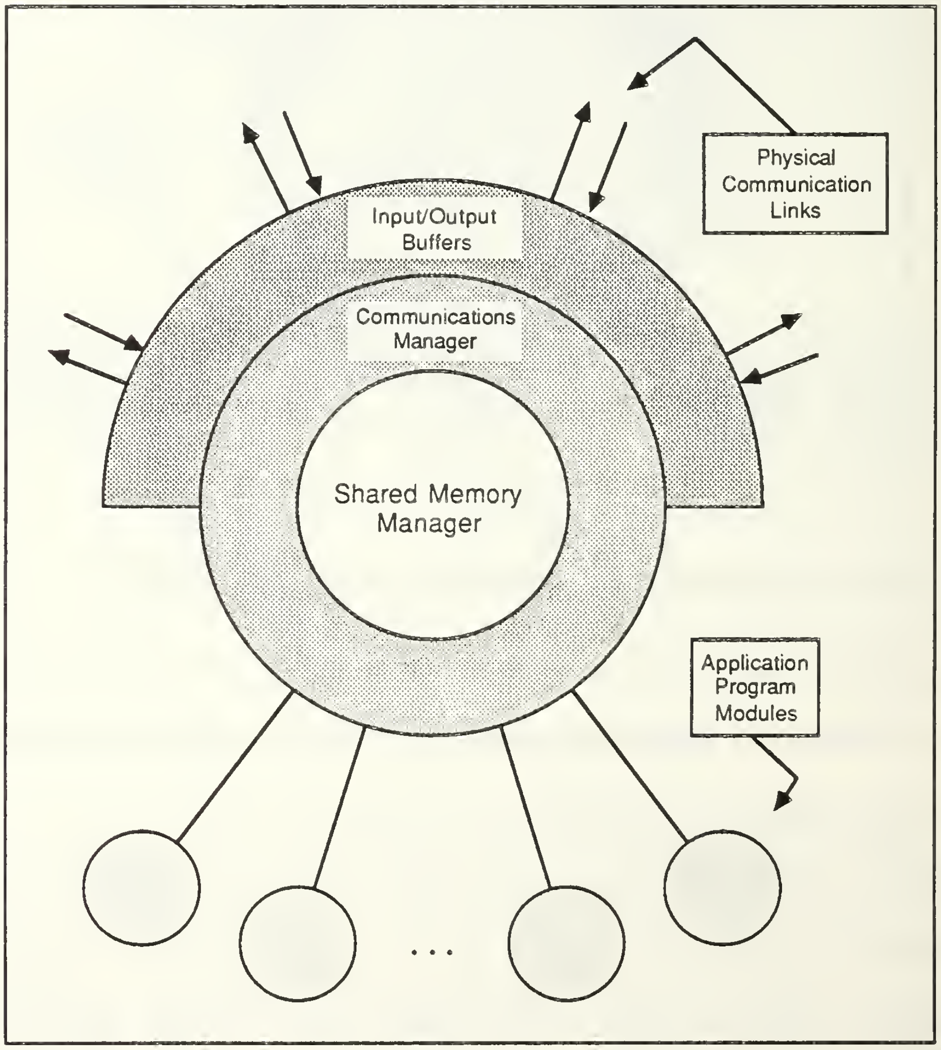


Figure 4.3. **Shared Memory Kernel Logical Block Diagram**

a. Input/Output Buffers

A set of input and and a set of output buffers are provided for the hardware communications links. These buffers decouple the relatively slow link data transfers from the operation of the kernel. The buffers enable the kernel to operate in parallel with network data transfers.

b. Communications Manager

All communications between a kernel and the library interface procedures and between kernels on different nodes are formatted as packets. Two different packet formats are used. One format is used for communication between a kernel and a library interface procedure and one is used for communication between kernels on different nodes in the network. The communications manager perform any necessary conversions between the two packet formats. Figure 4.4 depicts an example of each type of communications packet. A listing of all the packet types is included in Appendix C.

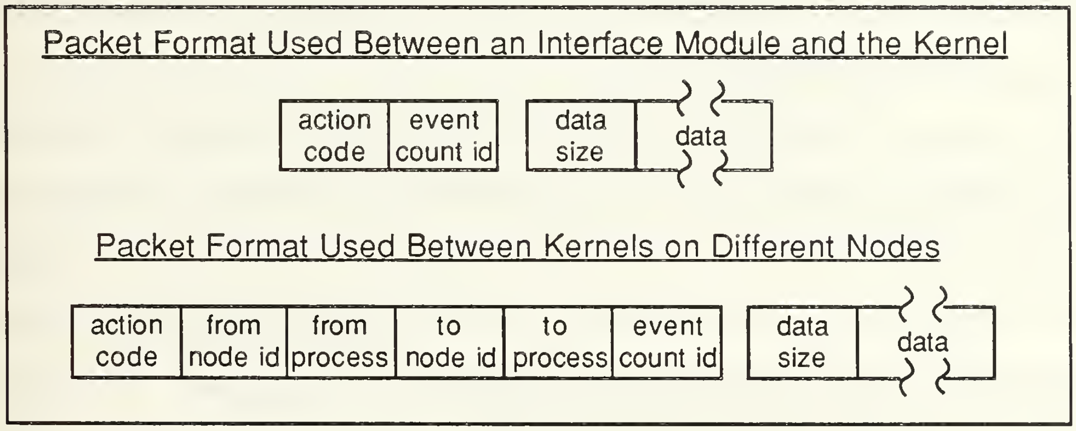


Figure 4.4. **Kernel Communication Packet Formats**

The communications manager processes all received packets. If a packet destination is a remote node, the packet is routed to the node via the hardware communications link identified by the node.link data structure. If the packet is for a local application program module interface procedure, the packet is routed to that interface procedure via the appropriate local link. Otherwise, the packet is processed by the node's shared memory manager.

c. Shared Memory Manager

The shared memory manager actually performs the operations defined by the various library interface procedures. As a result of external requests, this module accesses the kernel data structures and, if a response is required, generates appropriate message packets for return to the requesting process.

2. Data Structures

Operation of the kernel is best described by examining the set of data structures that support the kernel. These data structures are central to operation of the kernel. Diagrams of these data structures and their interrelation are shown in Figures 4.5, 4.6, and 4.7.

a. node.link

This array is used to represent the physical configuration of a network. Each node in the network is assigned a unique identifying number. By convention, the assigned numbers start at zero and are consecutive. The node.link data structure is a one-dimensional array with one element for every node in the network. The value of the ith element in the array identifies the number of the hardware link that is used to send messages to the network node with identifying number i. This does not necessarily mean that the ith node is at the other end of the specified link. Rather, it means that the node on the other end of the specified link is the next node in the path to the ith node.

Note that the node.link array may be viewed as one row extracted from a type of adjacency matrix that defines the network. Such an adjacency matrix array is, in fact, used at compile time to define the individual node.link arrays. As a result, each kernel hold a portion of the overall matrix. Currently, the programmer defines the contents of this matrix as the last step of the software development process.

b. count.node

As was done to uniquely identify nodes, each event count and sequencer in the system is assigned a unique identifying number. Again, by convention, these numbers start at zero and are consecutive. The count.node structure is a one-dimensional array with one element for each event count and sequencer in the system. The value of the ith element in the array identifies the number of the node that maintains the count assigned identifying number i. This array identifies which processor in the network is responsible for maintaining each event count.

c. count.array

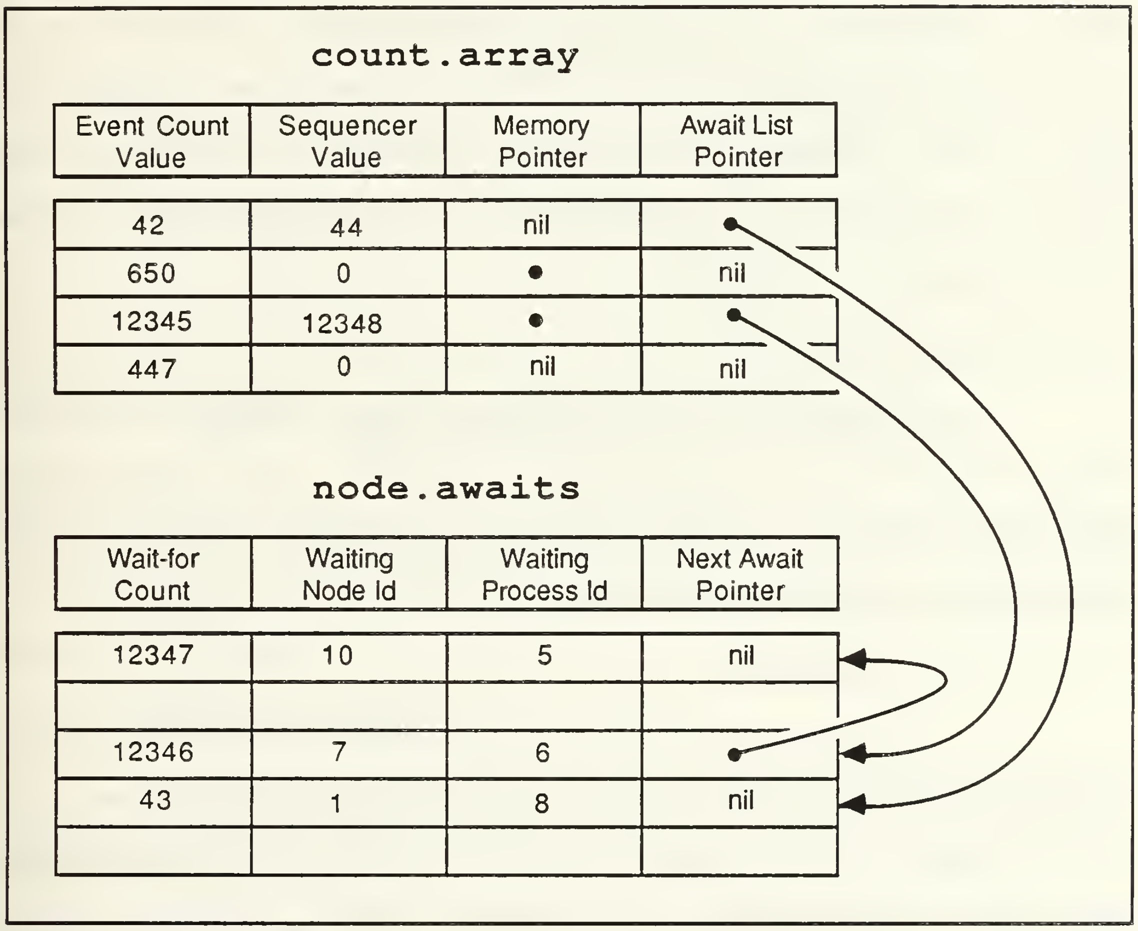
This is a two-dimensional array that contains four elements for each event count and sequencer maintained by a kernel at a node. The first two of these elements are the current values of the event count and sequencer. The third element is either a nil pointer or a pointer to the base of a shared memory segment associated with the event count and sequencer. The fourth element is either a nil pointer or a pointer to a list of processes which have been suspended by executing the await library procedure. This list is ordered by value of the count value specified when a processes executed the await procedure. Each event count and sequencer has a separate waiting process list associated with it.

d. count.array.indices

This is a one-dimensional array that has one element for each event count in the system. For counts maintained at a particular node, this array contains the index of that count's data in the count.array data structure. For counts not maintained at the node, the array contains the nil token. Use of this array speeds access to a count's data in the count.array data structure by providing a direct pointer to the desired row and avoiding having to search through the count.array to find the correct row.

e. node.awaits

As was mentioned earlier, associated with each event count and sequencer is a list of suspended processes waiting for particular count values to be reached. These lists of suspended processes are stored using the node.awaits data structure. The data structure is a two-dimensional array where each row of the array represents an entry for one suspended process. There are four data elements for each entry. The first element of the entry is the count value for which the suspended process is waiting. The next two elements identify which process at which node has been suspended. The final element is either a nil pointer or a pointer to the next entry in that particular waiting process list. These pointers and the waiting process pointers in the count.array data structure are row index values of the node.awaits data structure. Figure 4.5 shows an example waiting process list.



count.array

Event Count Value	Sequencer Value	Memory Pointer	Await List Pointer
42	44	nil	●
650	0	●	nil
12345	12348	●	●
447	0	nil	nil

node.awaits

Wait-for Count	Waiting Node Id	Waiting Process Id	Next Await Pointer
12347	10	5	nil
12346	7	6	●
43	1	8	nil

Figure 4.5. **Kernel Waiting Process List Structure**

f. free.list

This is a one-dimensional array that has one element for each array position in the node.awaits data structure. The free.list data structure holds a list of node.await data structure row indices that are not in use and may be allocated to any event count and sequencer's waiting process list. As entries are removed from wait process lists, the associated node.awaits row indices are returned to this free list.

A pointer into the free.list data structure identifies the value of the next available node.awaits row index. When this next index is allocated, the pointer is incremented to find yet the next available node.awaits row index. When an index is returned to the free.list data structure, the pointer is decremented and the returned index is stored. Figure 4.6 illustrates the use of the free list and associated pointer.

g. count.size

This is a one-dimensional array with one element for each event count and sequencer in the system. The value of the ith element in the array identifies the number of bytes of shared memory that are to be associated with count i.

h. node.data

This array is the block of shared memory available at a node. Segments of this memory block are apportioned based on the values in the count.size data structure for counts being maintained at that node. The memory pointers in the count.array data structure

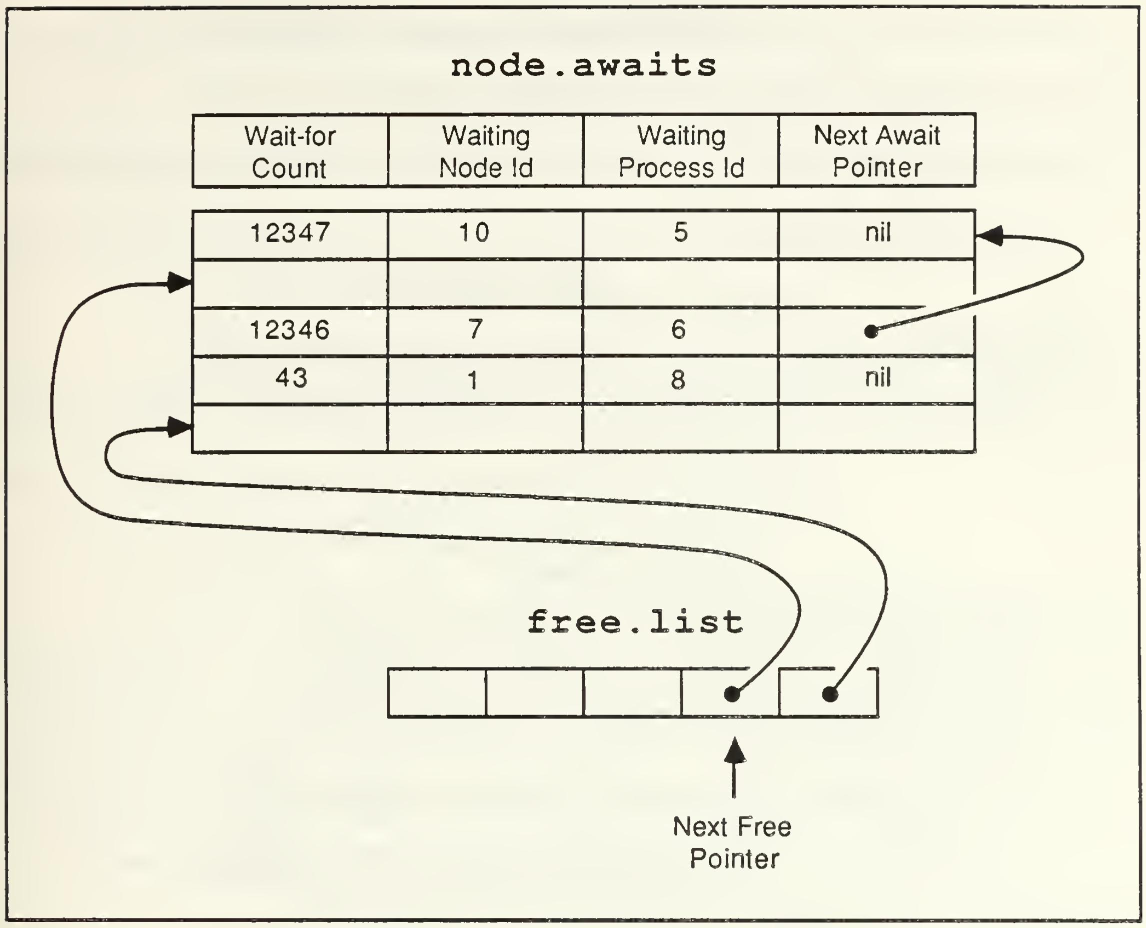


Figure 4.6. **Kernel Waiting Free List Management**

point to the start of individual shared memory segments in this block of memory. Figure 4.7 shows how this shared memory array is configured and accessed.

3. Library Procedures

A functional description of each of the library interface procedures is provided below. Each procedure includes a reference to a "link" parameter. This parameter is a bidirectional communication channel that links or connects a distributed application program module to the kernel. The program module does not directly use the channel for any communication. The library interface procedures use this channel internally for communicating with the kernel. The detailed source code for the procedures is provided in Appendix C.

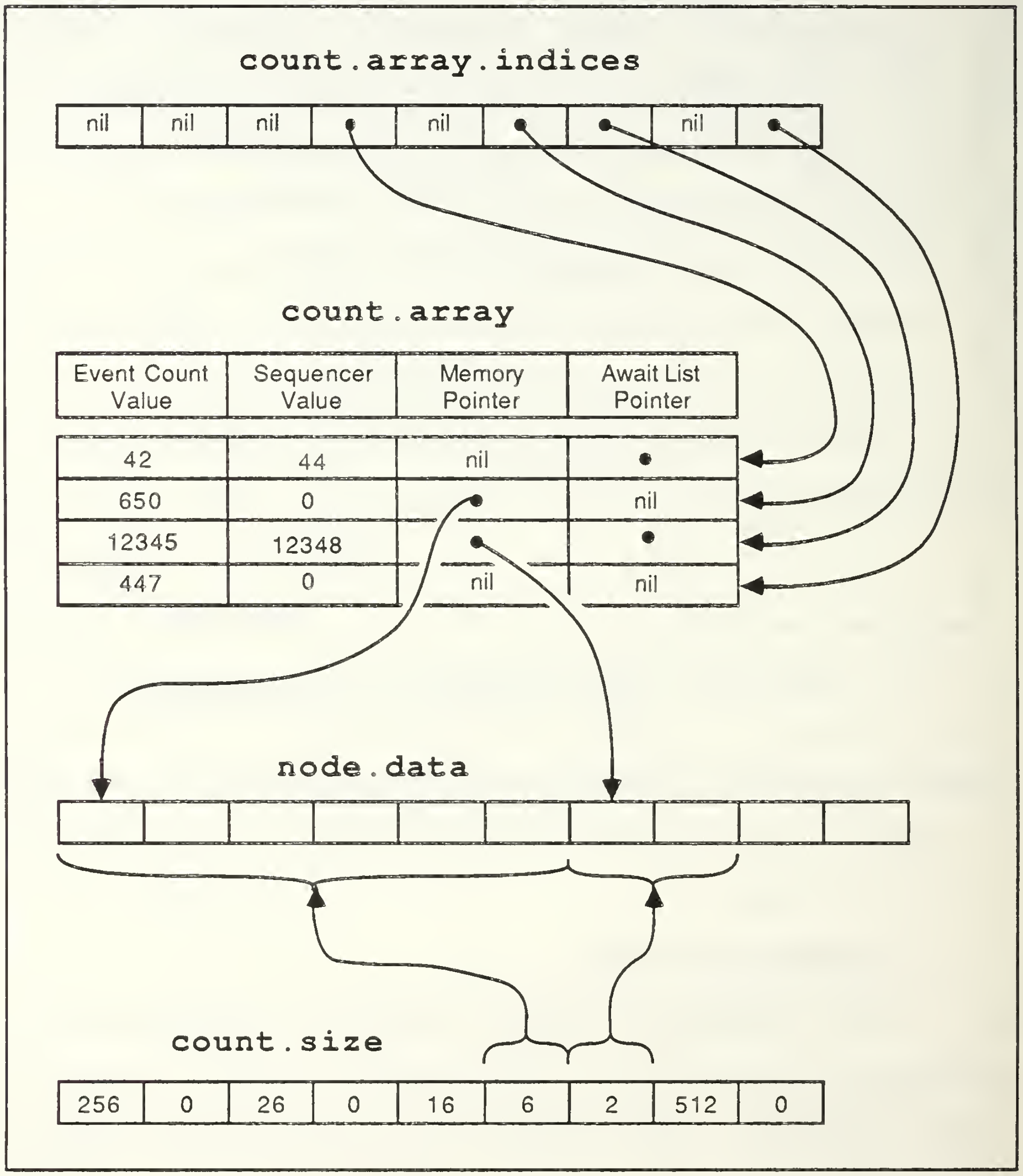


Figure 4.7. **Kernel Shared Memory Access Data Structures**

a. **read(link, count.id, count.value)**

This procedure performs the read operation on the specified event count. The value of count.value is set to the current value of the count.

b. **advance(link, count.id)**

This procedure increments the value of the specified event count. If other processes are waiting on the new value of the event count, these waiting processes are resumed.

c. **await(link, count.id, count.value)**

The executing process is suspended until the value of the specified count reaches the argument count.value. If the value of the specified count is already greater or equal to the argument count.value, the process executing the await call continues execution.

d. **ticket(link, count.id, ticket.value)**

This procedure sets the value of ticket.value to the value representing the next available reservation for the specified sequencer.

e. **put(link, count.id, index, byte.array)**

This procedure provides write access to the shared memory segment associated with the specified event count. The array of bytes passed to the procedure is stored in the shared memory segment offset from beginning of the shared memory segment by the number of bytes specified by the value of the index parameter.

f. get(link, count.id, index, byte.array)

The get procedure complements the put procedure by providing read access to the shared memory segment associated with the specified count. The array of bytes passed to the procedure is read from the shared memory segment offset from beginning of the shared memory segment by the number of bytes specified by the value of the index parameter. The number of bytes read is based on the size of the byte.array parameter.

g. ecs.kernel(link.array,count.node,count.size,node.id, node.link)

The procedure for the kernel is itself included in the library. This procedure is executed on each node in the system. The link.array parameter for the kernel is an array of all the individual links that connect the application program modules at a node to the kernel. The other kernel parameters are described in detail in the data structures section of this chapter.

D. PROGRAMMING

This section presents a brief overview of how to program using the shared memory interface. A detailed programming example is presented in Appendix D. Programming using the shared memory interface is accomplished in three basic steps.

The first programming step is to divide the program into modules that should operate in parallel and to define the shared memory segments and event counts and sequencers that will be required for communication between and synchronization of the modules. This step should performed without considering the physical configuration of a network.

The next step is to code the individual modules using the library interface procedures to manipulate the event counts and sequencers and to access the shared memory segments. Since this step is also accomplished without considering the physical configuration of the network, the modules may be coded independently.

The final step of the software development process is to apportion modules and shared memory segments amongst different processors in a network. Although any apportionment of the modules and memory segments is logically equivalent, this placement process may be influenced by practical considerations. For example, one would not want to place all the computationally intensive modules on the same processor. Further, it would seem reasonable to locate modules sharing the same memory segment physically close to the network location of the memory segment. To help quantify the factors that may influence the network placement of modules and shared memory segments, the following section evaluates the performance of the shared memory interface under a variety of conditions.

E. EVALUATION

To evaluate the prototype programming interface, a set of tests was conducted using the interface. The objective of this testing was to provide a representative measure of the communications performance of a network when using the programming interface. It was desired to determine how the network communications performance was affected by variations in certain parameters. This testing utilized T414 Transputers operating with a clock speed of 15 MHz and a link speed of 20 Mbits per second. These tests are described in the following paragraphs. It should be noted that the prototype programming interface was not optimized for maximum performance. Because of this, the interface performance documented here can likely be improved. Chapter VI describes some ways in which the programming interface performance can be improved.

1. Basic Interface Procedure Timing

As the first step in examining the performance of the programming interface, it was chosen to measure the execution times of the various interface procedures. A test program was developed to execute each of these procedures in a loop. The time required to execute the loop with each of the interface procedures was measured. The time required to execute the loop without an interface procedure (i.e., a "null loop") was measured and subtracted from the interface procedure loop times. The resulting time was used to determine the average time required for a single interface procedure execution.

Interface procedure execution times were measured under several sets of conditions. Specifically, the network distance between an interface procedure and its target event count and sequencer was varied and, for the put and get operations, the size of the argument data element was varied. The results of this testing are shown in Figures 4.8, 4.9, and 4.10.

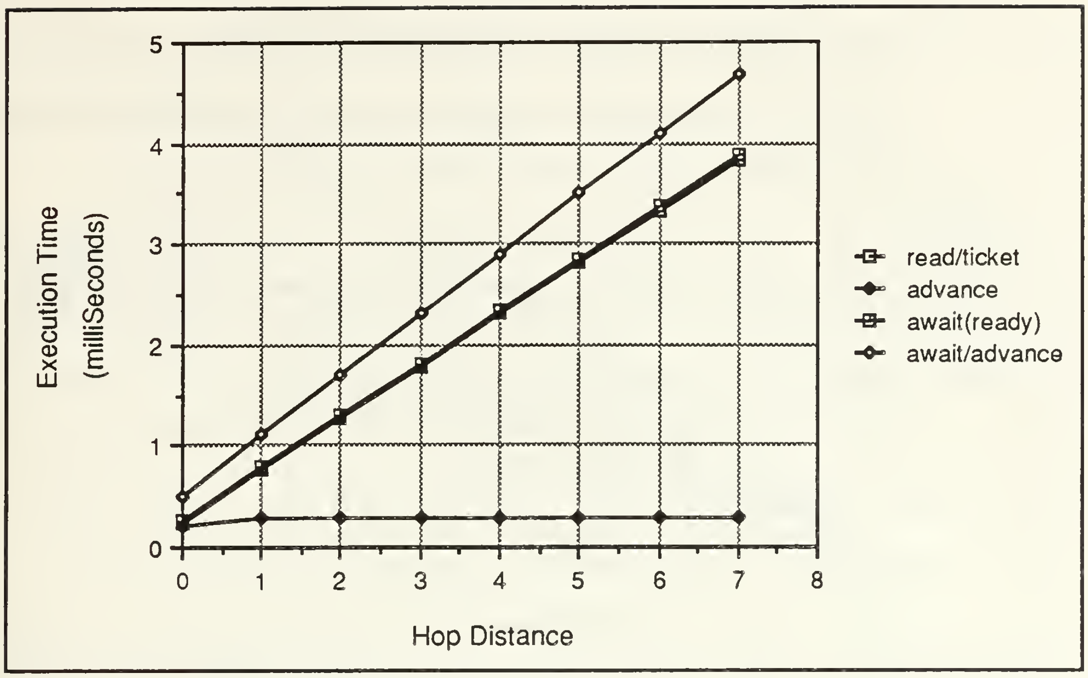


Figure 4.8. **Primitive Operation Timing Test Results**

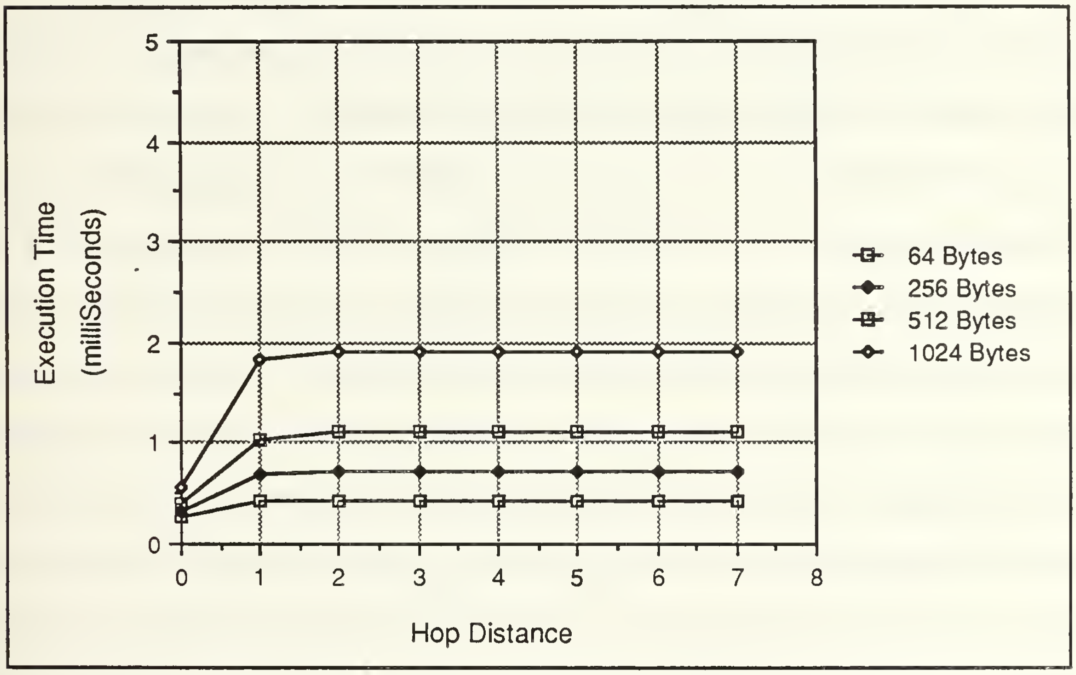


Figure 4.9. **Put Operation Timing Test Results**

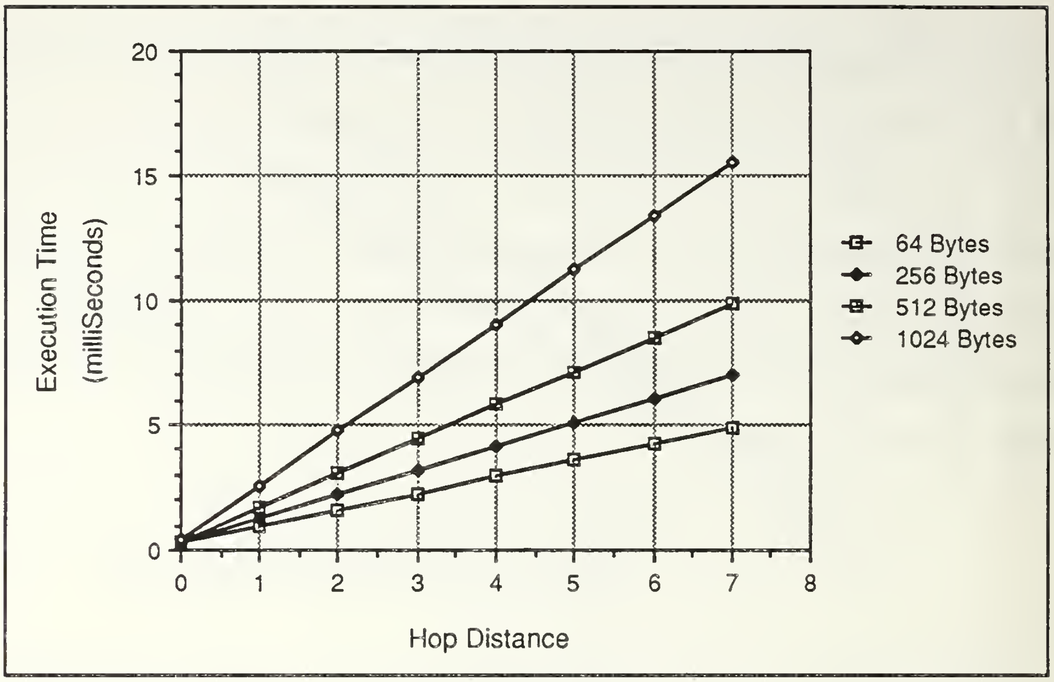


Figure 4.10. **Get Operation Timing Test Results**

Figure 4.8 shows two basic types of execution behavior. First, the execution times for one group of interface procedures is relatively constant over the range of conditions tested. This group of procedures includes those that do not require any response from the distributed kernel. The procedure can pass its communications packet to the kernel and the application program module can proceed without waiting for a reply. Since communication between the interface procedure and the kernel is via memory-to-memory transfer, the transmission of a communications packet is fast and relatively insensitive to variations in data element size over the range of kernel operation. Note also in Figure 4.9 that the execution time for the put operation is significantly less for the zero-hop (same node) case than for the one-hop case. This is because the zero-hop case is executed by performing a memory-to-memory transfer of data as opposed to the slower physical link transfer of data in the one-hop case.

The second type of behavior is for interface procedures that do require a response from the distributed kernel. As can be seen in Figure 4.8 for the read, ticket, and await procedures and in Figure 4.10 for the get procedure, execution times are strongly dependent on the test conditions. As the network distance is varied, the overall execution time increases because the "round trip" communication time in the network increases. Further, since data transfer between nodes via the hardware links is significantly slower than the memory to memory transfer, the effects of data size on execution time for the get procedure become significant.

The results of this testing imply that there is a preferred location for the counts and shared memory segment for a producer and consumer of data located at different nodes in the network. Since the put operation is relatively insensitive to network distance and since the get operation is greatly affected by network distance, it would appear that the shared memory segment for a producer consumer pair should be located at the consumer's node. A test was performed to confirm this. In this test, the count and shared memory segment for a producer and consumer pair were placed at the producer's node, then at the consumer's node. The resulting data communication rates were measured for various data element sizes. The results of this testing are shown in Figure 4.11 for a network distance of seven nodes. The results of this test show that the highest communications rate is obtained when the shared memory segment and associated event count for a producer/consumer pair is located at the consumer's node.

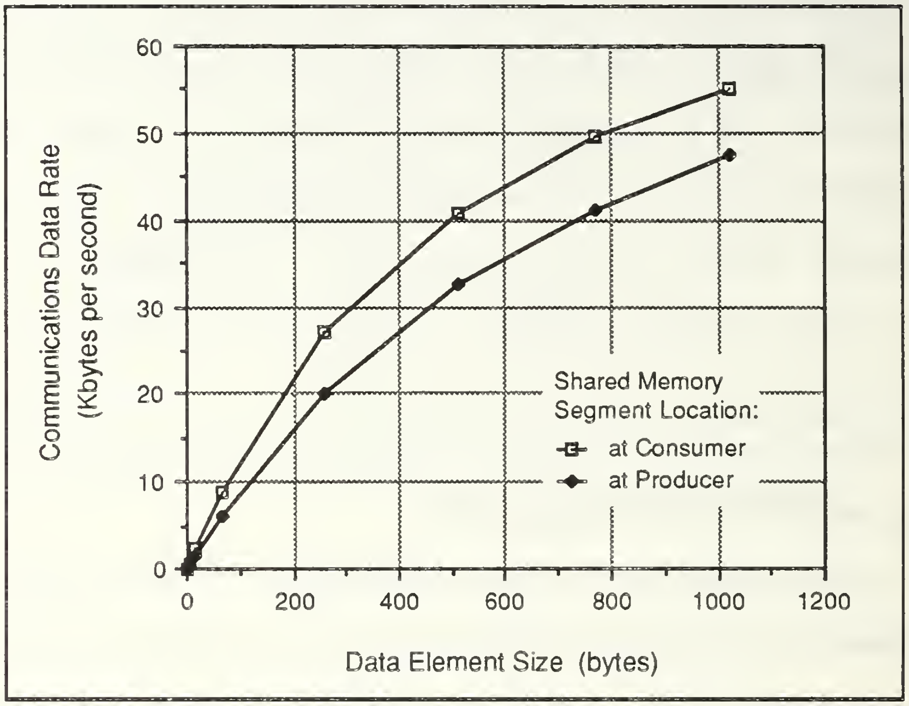


Figure 4.11. **Count and Shared Memory Location Effects on Communication Rate**

2. Node Distance Effects on Communication Rate

The distance between an application program module producing data and the application program module consuming that data can be measured as the number of physical links or "hops" in the communications path between the node locations of the modules. A test was performed to determine the effect that hop distance could have on the overall data communication rate between application program modules. In this test, a producer and consumer of data were separated by increasing hop distances. The library interface procedures were used to transfer data between the producer and consumer at the maximum possible rate. In addition, since the size of a communication message has a demonstrated effect on performance (Chapter III), the size of the data element being communicated using the programming interface was varied. The resulting communications data rates are shown in Figure 4.12.

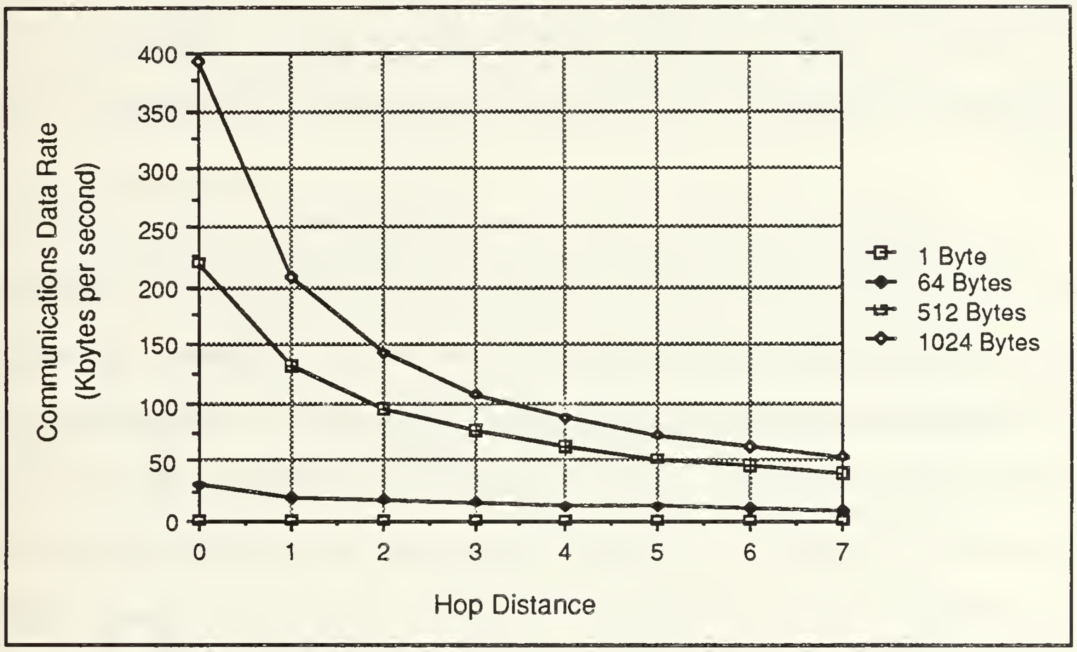


Figure 4.12. **Hop Distance Effects on Communication Data Rate**

This graph shows that as the hop distance increases, the maximum data rate decreases. This decrease is rapid for first few hops but less significant as the hop distance is further increased. This is because each additional hop represents a proportionally smaller increment of in-line delay to the data transfer. As was shown in

Chapter III for the Transputer in general, communication of larger data elements is more efficient when using the programming interface.

Processes executing on the nodes between the producer and consumer may also be affected by the data transfer through a node. To examine this, a looping calculation was initiated on a node immediately between a producer and consumer pair. The number of loops executed per unit time during a data transfer was measured and compared to the no-data transfer looping rate. Again, varying sizes of data elements were used. The results of this testing are shown in Figure 4.13.

Figure 4.13 shows that lowest performance of about 70 percent of normal occurs when using the smallest-sized data elements. As the size of the data element increases, the performance increases to a maximum of about 80 percent of normal. Performance of the intermediate process improves when the data element size is increased because the overhead associated with communicating a single message becomes relatively smaller. This is the same general effect as was shown in Chapter III for communication with varying sized data packets.

3. Hardware Link Sharing Effects on Communication Rate

When using the programming interface, it is likely that data communication between several pairs of producers and consumers will be conducted using the same hardware communications link. In this case, the kernel in the node on either end of the link is also involved in the communication. A test was performed to determine what effects this link sharing had on the overall data communications rate between the nodes. In this test, producers and consumers were placed on adjacent nodes. Producers were placed on each node in the same number to provide balanced bidirectional utilization of the hardware link. The number of producers on each node and the size of a communication data element was varied. The results of this test are shown in Figure 4.14. This plot shows that, although some degradation in communications performance does occur due to kernel overhead, a substantial data rate can be maintained when a hardware link is shared between several users.

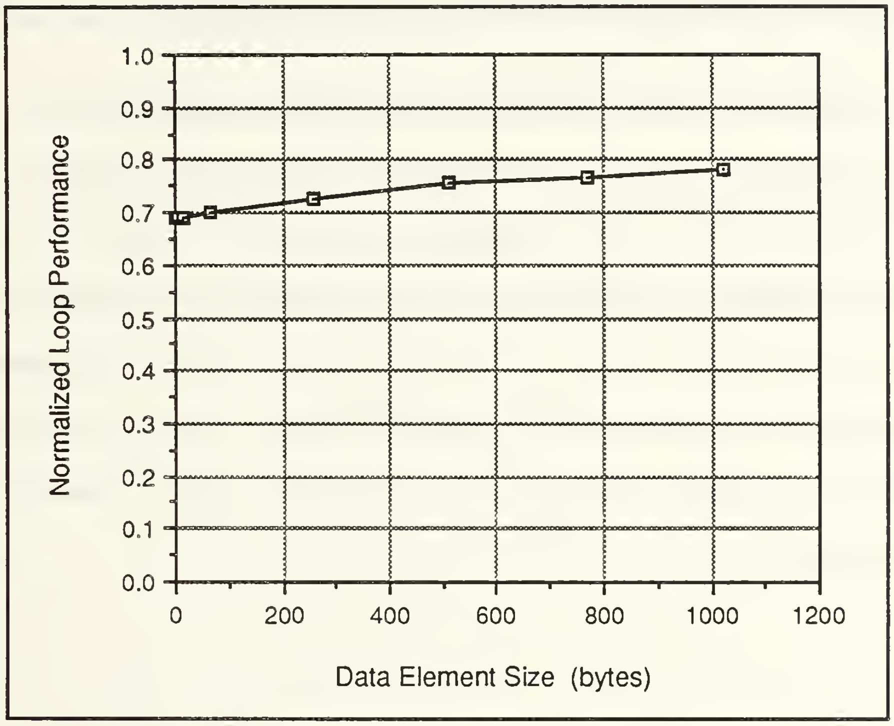


Figure 4.13. **Intermediate Process Degradation During Communication**

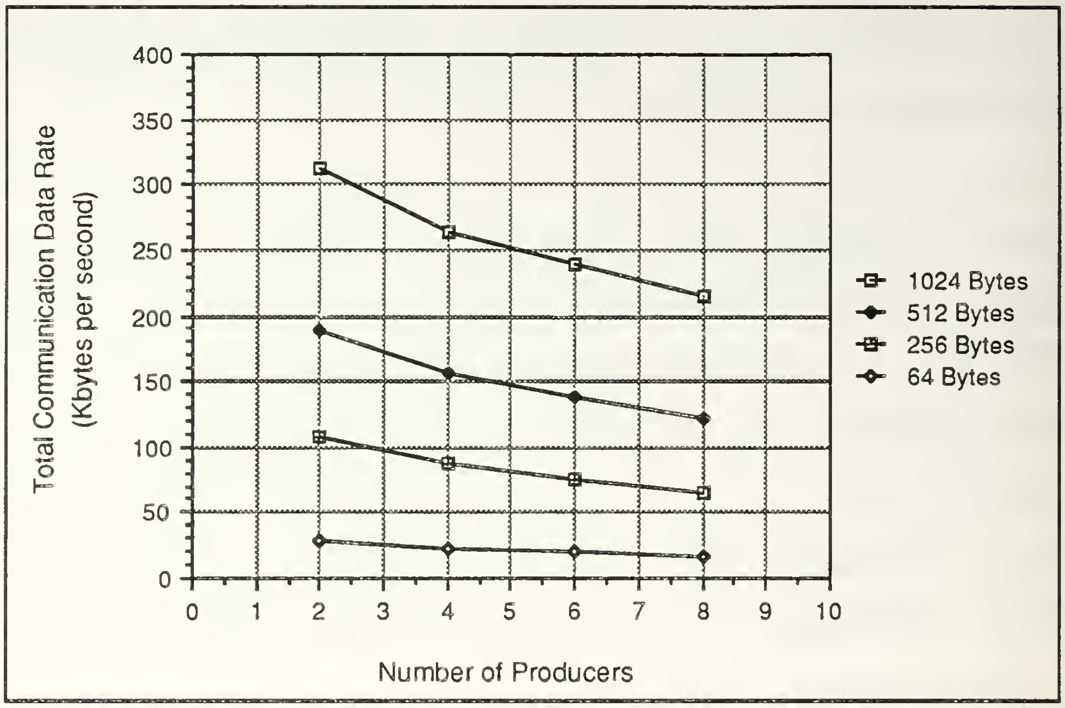


Figure 4.14. **Shared Hardware Link Communications Rates**

4. Multiple Link Effects on Communication Rate

In most cases, several links on each Transputer in a network will be connected and utilized at any one time. To test the communications performance of the kernel under these conditions, a varying number of hardware links on one Transputer were bidirectionally loaded. The resulting data rates were measured for different data element sizes. The results of the test are shown in Figure 4.15.

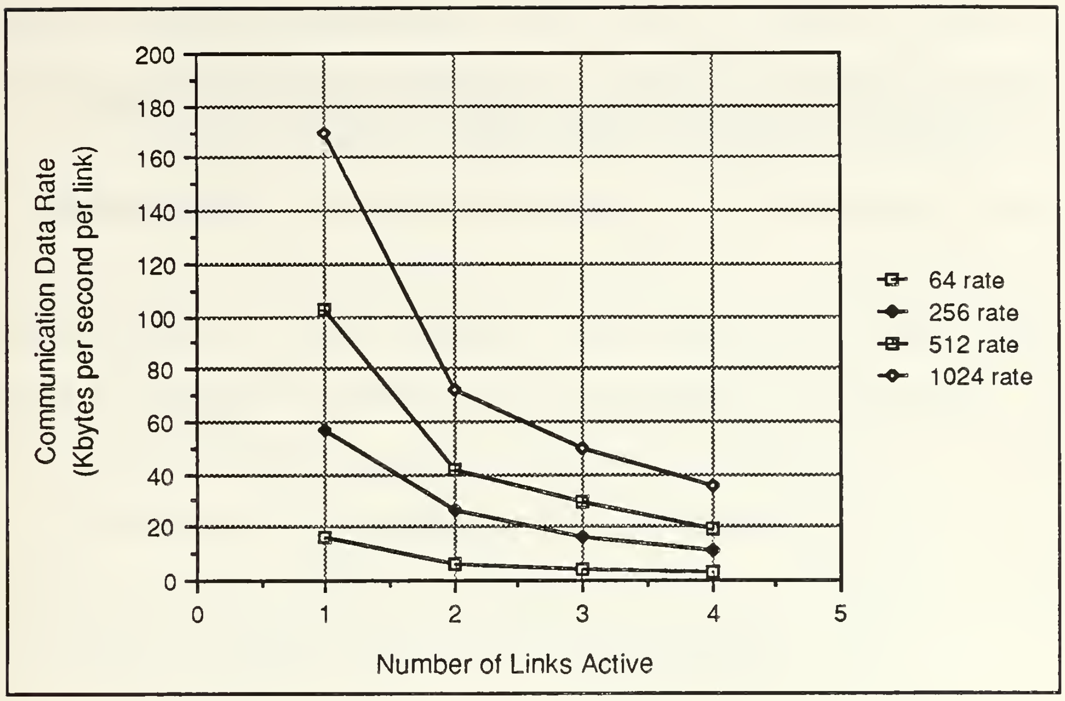


Figure 4.15. **Multiple Hardware Link Communications Rates**

Figure 4.15 shows that, as the number of links used increases, the data actually communicated via each link decreases. However, in Chapter III it was shown that link communication rate was essentially independent of the number of links active. The kernel must, therefore, the cause of this reduced link utilization. Since all data messages must be processed by the kernel and since the kernel can only process and transfer a fixed maximum number of messages in any time period, a limit on the data rate through the kernel must exist. Since Figure 4.15 shows the communication rate per link decreasing by more than a factor of two when a second link is activated, it appears that the kernel data rate limit is less than the capacity of a single communication link. Activating additional links, therefore, except to reduce the hop distance between nodes in a network, does not increase the overall communication bandwidth of the network. Chapter VI discusses potential changes to the kernel which may improve the programming interface performance.

V. MESSAGE-PASSING MODEL PROGRAMMING INTERFACE

A. BACKGROUND

The shared memory model is one method for isolating the distributed systems software designer and programmer from having to consider the physical configuration of a network of Transputers. It is also possible, however, to utilize a message passing model to provide for this isolation. To investigate this alternative, a prototype programming interface based on message passing has been developed. This chapter describes and evaluates this prototype.

B. IMPLEMENTATION

Since the Transputer is based on the concept of Communicating Sequential Processes (CSP), using this concept as a basis for a message-passing prototype was a natural choice. Communications in the message-passing prototype are, therefore, based on CSP communications "rules." Specifically, message passing is logically point-to-point, synchronousm and unbuffered.

A significant goal in the development of this interface was to permit application program modules to be written in the Transputer's "native" high-level language, OCCAM, without requiring any additional external procedure references. In this way, existing modules programmed in OCCAM could be used with the interface.

As with the shared-memory model prototype, the core of the message-passing prototype is a distributed kernel that executes on each node in the network. Figure 5.1 shows the physical configuration of such a network. Note that this physical configuration is very similar to the physical configuration used in the shared-memory model prototype.

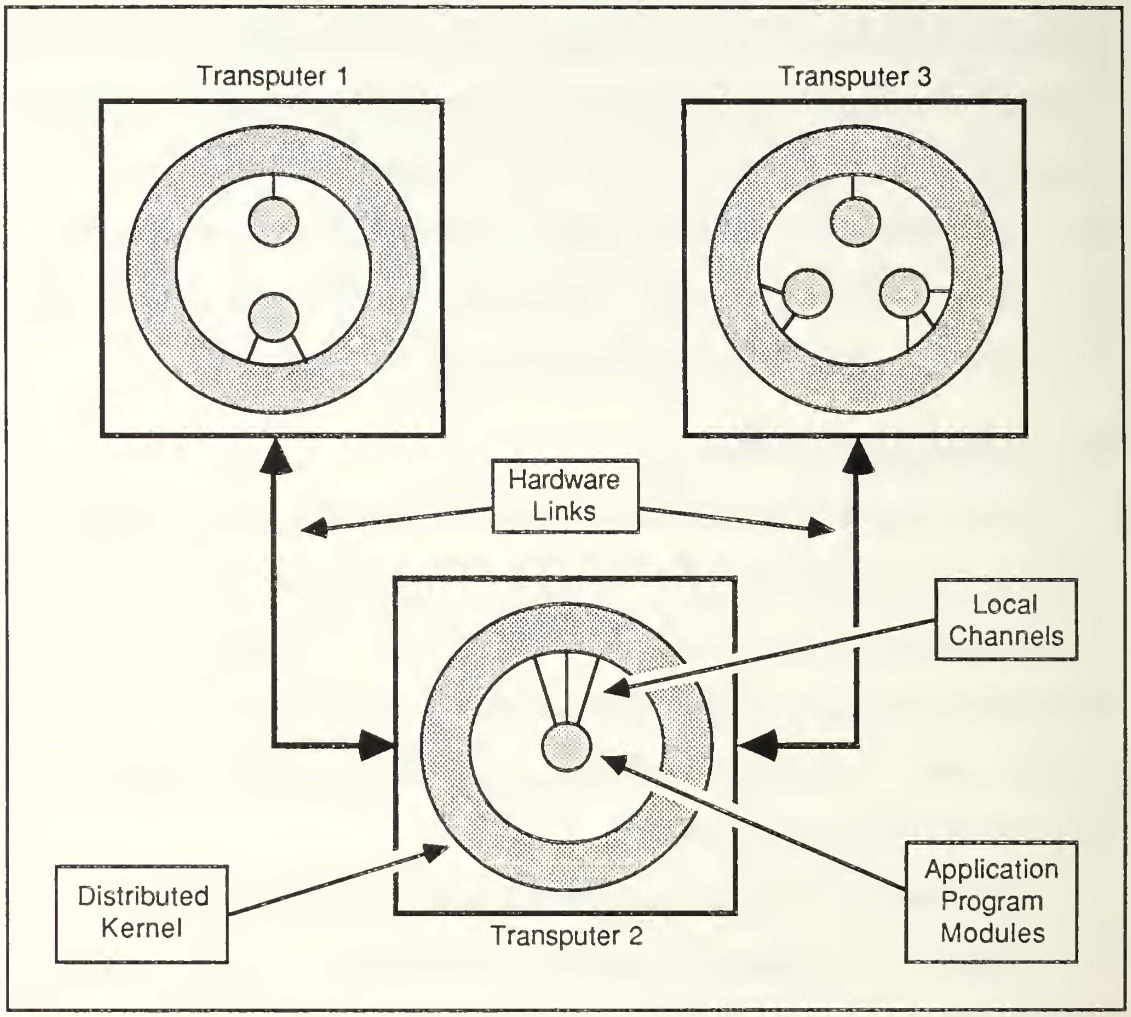


Figure 5.1. **Physical Configuration of Message-Passing Kernel**

Figure 5.2 depicts the logical configuration of the message-passing network as seen by the application. The logical configuration of the message-passing network shows individual application program modules interconnected by a network of global message-passing channels.

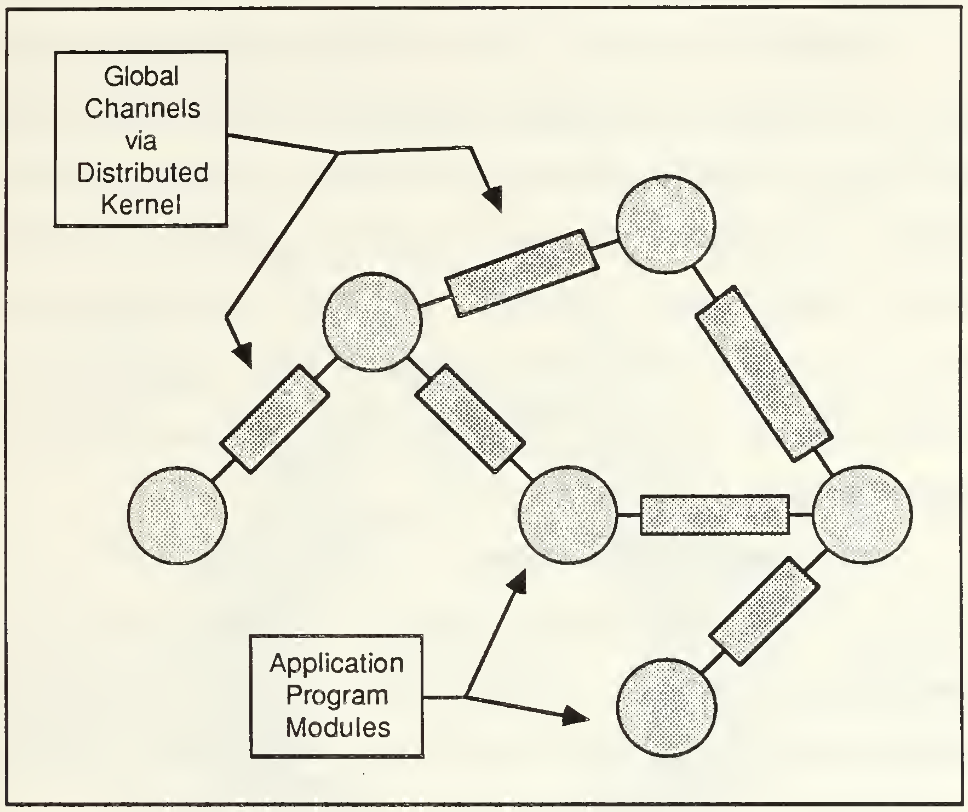

Global Channels via Distributed Kernel
Application Program Modules

Figure 5.2. **Logical Configuration of Message-Passing Kernel**

This chapter frequently uses the terms “local channel” and “global channel” when referring to the operation of the message-passing prototype. A local channel is an actual communications path that exists between a process and a kernel. Figure 5.1 shows several examples of local channels. Global channels, as shown in Figure 5.2, are the virtual communication paths that exist between two processes. These virtual communications paths are established and maintained by the distributed kernel. Local channels may be thought of as connecting processes to the ends of global channels.

1. The Kernel

Figure 5.3 shows a block diagram of the message-passing kernel. As shown, the kernel consists of three major parts: the Input/Output Buffers, Channel Controllers, and the Communications Manager. In addition, the kernel performs some initialization of the network. Each node broadcasts to the network a list of the global channel ends that it has been assigned. This information is used by the kernel to create a global routing table for network communications.

a. Input/Output Buffers

These buffers are identical to those used in the shared memory model prototype. They decouple the relatively slow link data transfers from the operation of the kernel. The buffers enable the kernel to operate in parallel with network data transfers.

b. Channel Controllers

The kernel creates one channel controller process for each local channel at a node. The channel controller is the physical connection between a local channel and the kernel and is the logical connection between a local channel and the appropriate global channel. Communication over the global channel is controlled using a simple protocol which is managed by these channel controllers. When the receiving end of a global channel is ready to receive an application program module message, the receiving end channel controller transmits a "ready to receive" message to the sending end controller. The sending end controller is then released to accept the application's message and send it to the receiving application program module via the global channel.

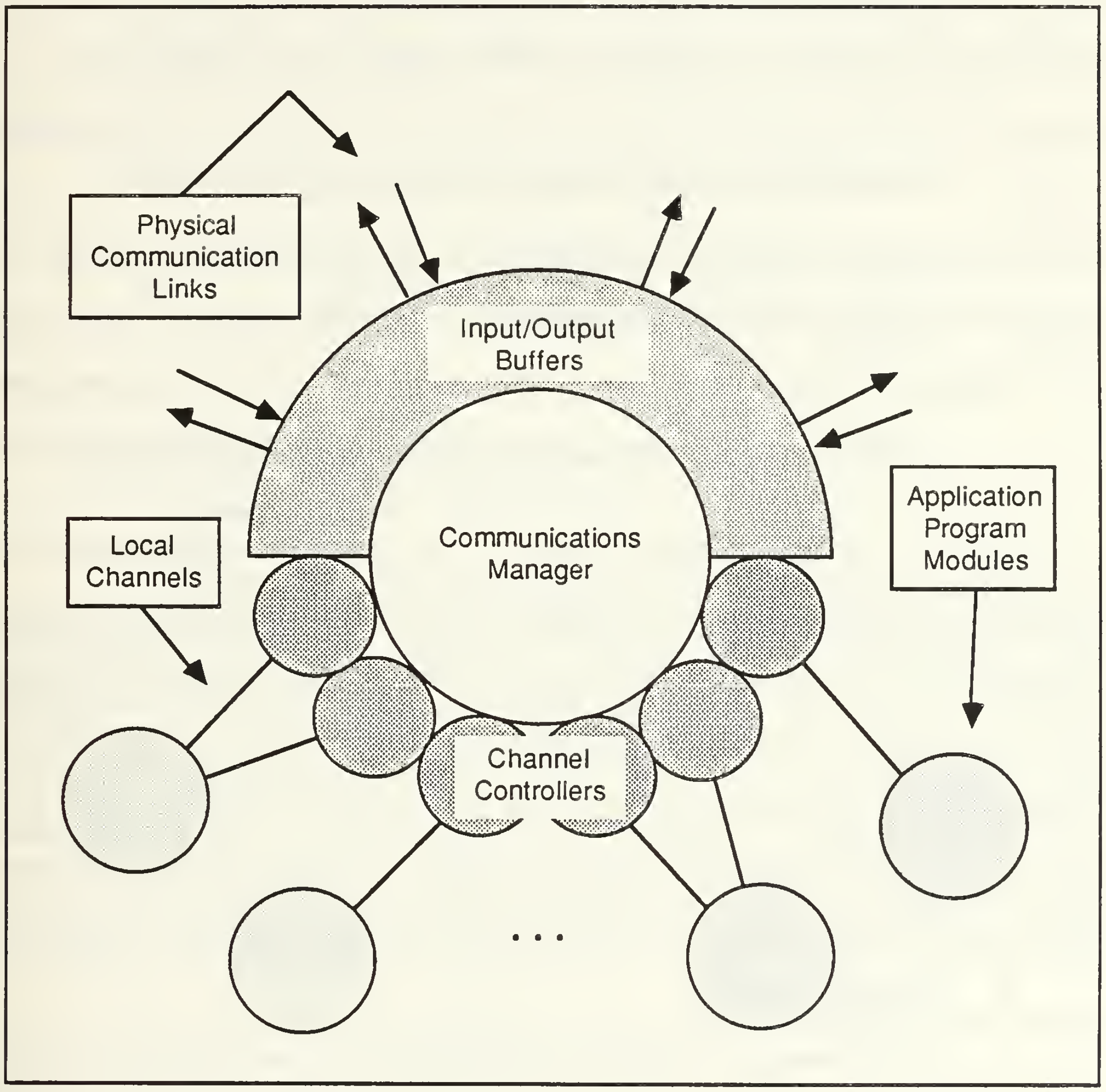


Figure 5.3. **Message Passing Kernel Logical Block Diagram**

c. Communications Manager

As in the shared memory interface, messages communicated over the network are formatted as packets. Figure 5.4 diagrams the packet format used. Each packet received from a hardware link or from a channel controller is processed by the communications controller and routed to its packet header identified destination.

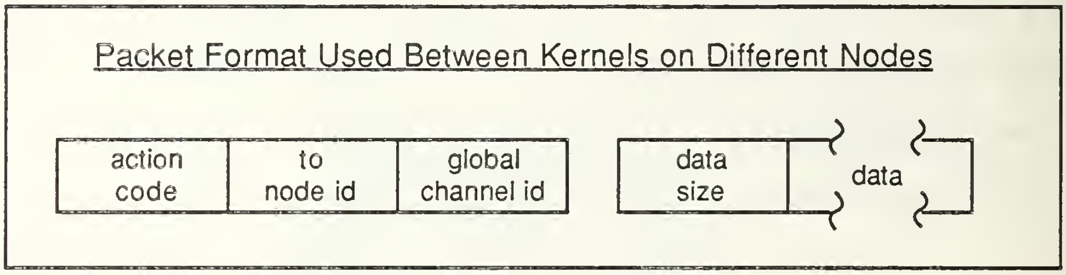


Figure 5.4. **Kernel Communications Packet Format**

At the current state of the kernel's development, messages transmitted from an application program module over a global channel are restricted to only one of the simple protocols predefined in the OCCAM language [PoMa87]. The available protocol provides for transmission of variable-length byte arrays (INT::[]BYTE). This is not a particularly limiting restriction since any structure in OCCAM can easily be RETYPED into an array of bytes for communications purposes.

2. Data Structures

The message-passing prototype kernel maintains a set of data structures that defines the characteristics of the physical and logical communications network. The overall operation of the kernel is fundamentally dependant on the interrelation of these structures. The kernel data structures are described in the following paragraphs. Figure 5.5 is an example of the manner in which these data structures are interrelated at two different nodes.

a. node.link

This data structure is identical to the node.link data structure used in the shared-memory model. The value of the **i**th element in the array identifies the number of the hardware link that connects to the next node in the path leading to node **i**. The array defines the node's view of a network. Currently, the programmer defines the contents of this data structure at the end of the software development process based on the particular network configuration that is to be used.

b. gchan.node

This structure is used to identify the end locations of network global channels. This is a two-dimensional array which has a two-element entry for each global channel defined in the distributed system. Each global channel in the network is assigned a unique identifying number. By convention, the assigned numbers start at zero and are consecutive. This array is indexed by a global channel's unique identifying number. The first element of an entry identifies the node location of the process that outputs to the global channel. The second element identifies the node location of the process that inputs from the channel. During kernel initialization, each node broadcasts a listing of global channel ends at the node. The kernel uses these

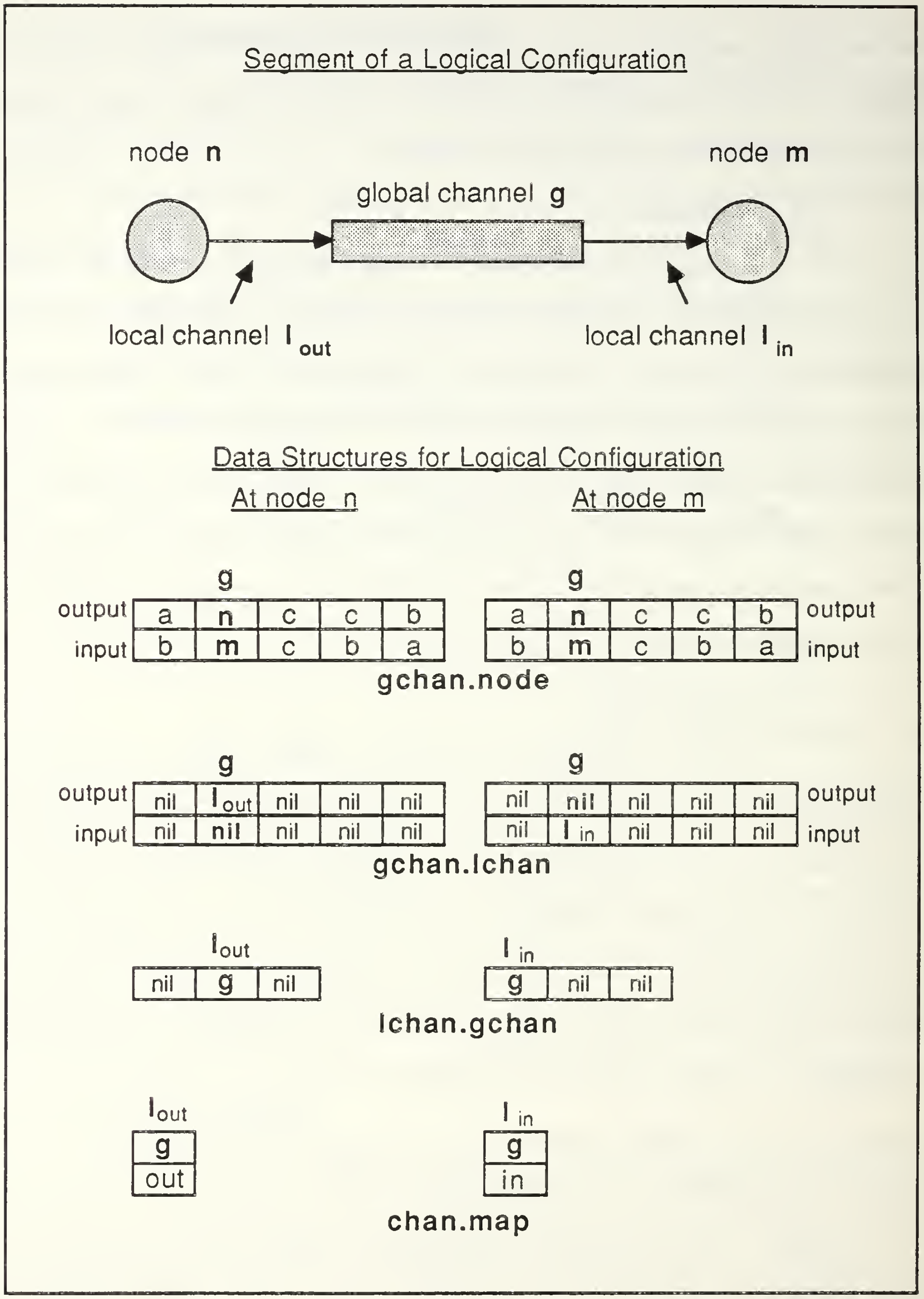


Figure 5.5. **Kernel Data Structure Interrelation**

broadcast messages to construct this array. As a result, each node in the system has an identical copy of this data structure.

c. gchan.lchan

If a global channel end is connected to a local channel at a particular node, this structure identifies which local channel the connection is to. This is also a two-dimensional array indexed by a global channel's unique identifying number. Each entry in the array consists of two elements. In general, the first element of an entry is the identifier of the local channel that outputs to the global channel; the second element is the identifier of the local channel that inputs from the global channel. At a particular node, this array only includes local channel identifiers for the global channels that output from or input to that node. All other entries in the array are nil at that node.

d. loc.chan

This data structure is an array of OCCAM communication channels. This array, potentially different at each node in the network, defines all the local channels that connect to global channels at a particular node.

e. lchan.gchan

This is a one-dimensional array with an element for each local channel at a node. The value of the **i**th element in the array is the identifier for the global channel to which local channel **i** is connected.

f. chan.map

Each node in a network has a different version of this data structure. The data structure is an array that holds two elements for each local channel defined at a node. For the ith channel defined in the loc.chan array, the ith entry in the chan.map array defines the end of the global channel "connecting to" that local channel. The first element of an entry is the unique identifier for the global channel. The second element of an entry identifies whether the local channel is connected to the sending or receiving end of the global channel.

2. Library Procedures

The Library for the message passing interface consists only of the distributed kernel procedure:

csp.kernel(node.id, node.link, chan.map, loc.chan).

This kernel procedure is executed on each node in the system. In general, the parameters for this procedure identify the physical and logical channel mappings for that node.

C. PROGRAMMING

This section presents a brief overview of how to develop a program using the message-passing interface. A detailed programming example is presented in Appendix F. Programming using the message-passing interface is accomplished in essentially the same three basic steps as is programming using the shared memory interface. The first step is to divide the program into modules that should operate in parallel and to define the point-to-point channels that will be required for communication between the modules. This modularization need not, however, consider the physical four-link limitation of an individual Transputer or the actual connectivity of a particular network.

The next step is to code the individual modules using OCCAM programming guidelines [PoMa87]. Thus far, the program development methodology is the same as would be used for developing any program in OCCAM.

The final step of the software development process is to apportion modules amongst different processors in a network. The same practical considerations that influence the placement of channels when using the shared memory model also need to be considered when using the message passing model.

D. EVALUATION

The same types of testing were performed for the message-passing interface as were performed for the shared-memory interface. This section provides the results of the message-passing model interface kernel testing and compares the test results of the two interfaces. The message-passing interface testing also utilized T414 Transputers operating with a clock speed of 15 MHz and a link speed of 20 Mbits per second. Since the message passing interface has no separate set of procedures for use in application program modules, separate interface procedure timing tests were not needed.

1. Node Distance Effects on Communications Rate

A test was performed to determine the effect increasing distance between nodes has on the overall data communication rate between application program modules. In this test, a producer and consumer of data were separated by increasing hop distances. Data messages were sent from the producer to the consumer at the maximum possible rate. The resulting communications data rates for different data message sizes are shown in Figure 5.6. Figure 5.7 provides a comparison of the maximum data communications rates for the two interfaces.

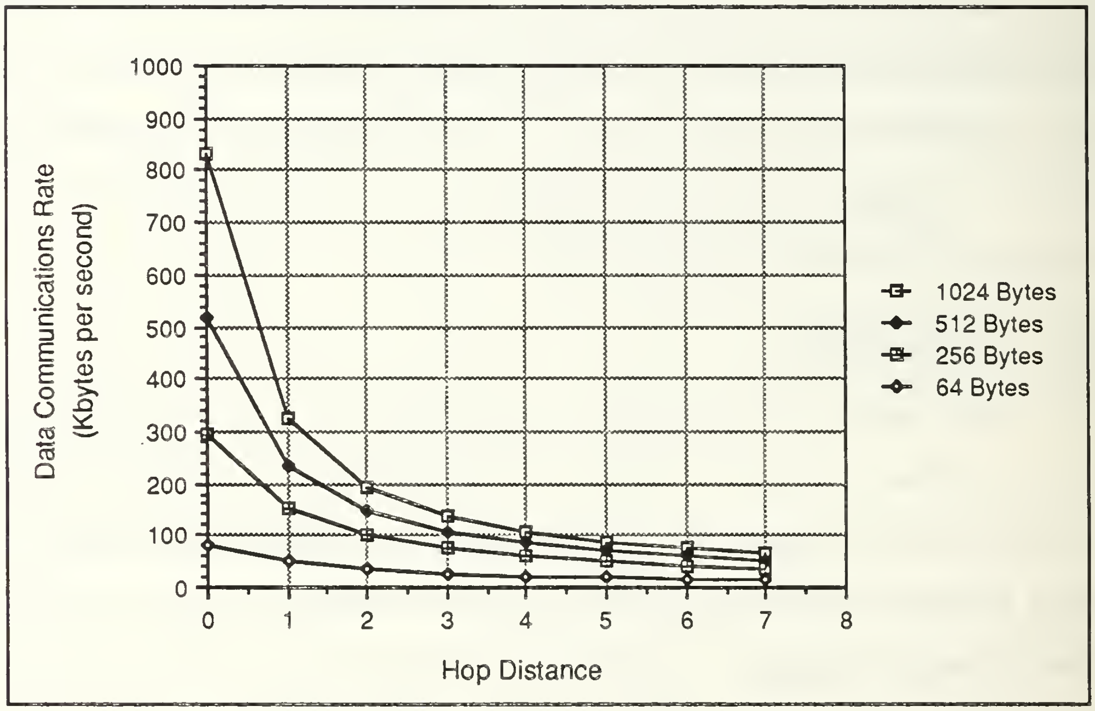


Figure 5.6. **Hop Distance Effects on Communications Data Rate**

Figure 5.6 shows that as the hop distance increases, the maximum data rate decreases. This decrease is rapid for first few hops, but less significant as the hop distance is further increased. This is because each additional hop represents a proportionally smaller increment of in-line delay to the data transfer. This same effect was shown for the shared memory kernel in Chapter IV.

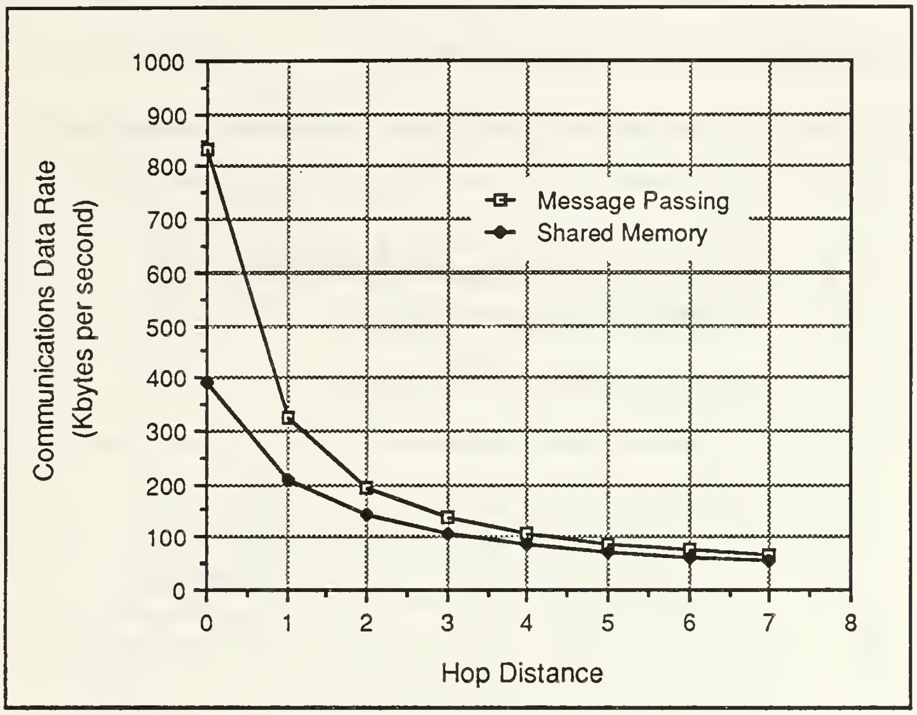


Figure 5.7. **Comparison of Hop Distance Effects**

Figure 5.7 shows that for smaller network distances, the message passing interface has a significant data communications rate advantage over the shared memory interface. However, as the network distance between the producer and consumer nodes increases, the communication performance difference between the two interfaces decreases. In both cases, minimizing the difference between the producing and consuming nodes improves performance.

With the message-passing model's greater data communication rate, it might be expected that performance of processes on nodes in the path between a producer and consumer would be degraded to a greater degree with the message-passing model than with the shared-memory model. Figure 5.8 shows the intermediate process degradation for both interfaces.

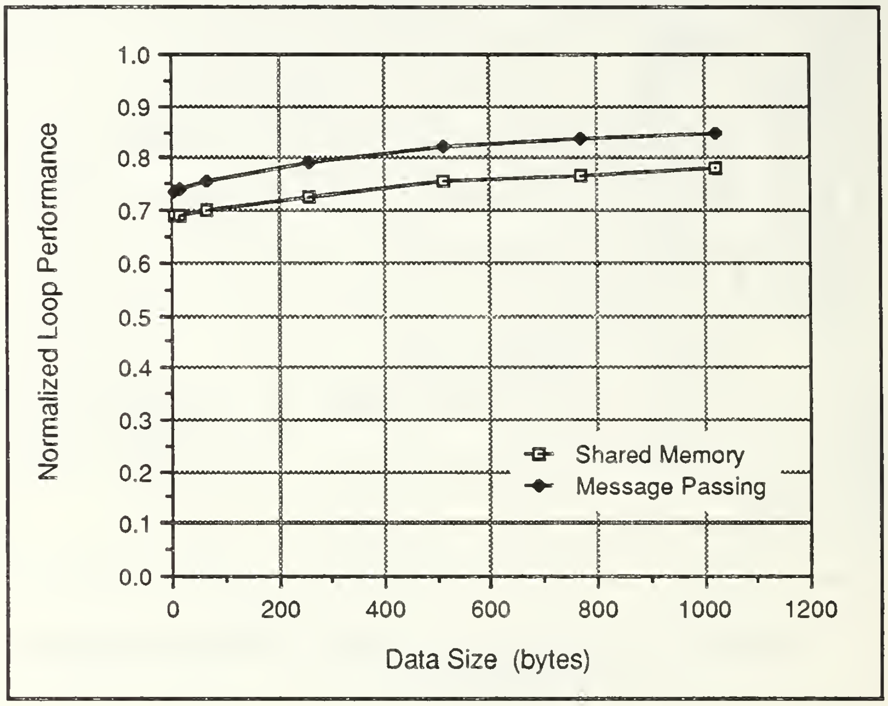


Figure 5.8. **Intermediate Process Degradation During Communication**

As can be seen, the degradation for the message-passing model is less than for the the shared-memory interface. The reason for this behavior appears to lie in the number of messages an intermediate node must process when using the two interfaces. In the shared-memory model case, each production/consumption cycle requires at least the transmission of three messages between nodes (an await message, an advance message, and either a put or a get message). The message-passing model only requires that two messages be sent (a receiver ready message and a data message). As shown in Chapter III, a greater degradation should be expected when using more messages to communicate a given sized data element.

2. Hardware Link Sharing Effects on Communication Rate

In the message-passing model, several global channels will likely use the same physical link for network communication. As with shared-memory interface testing, a varying number of producer/consumer pairs were executed on two adjacent nodes to examine the kernel's performance. The results of this test are shown in Figure 5.9. Despite some performance degradation, these results show that a link can be effectively shared between global channels.

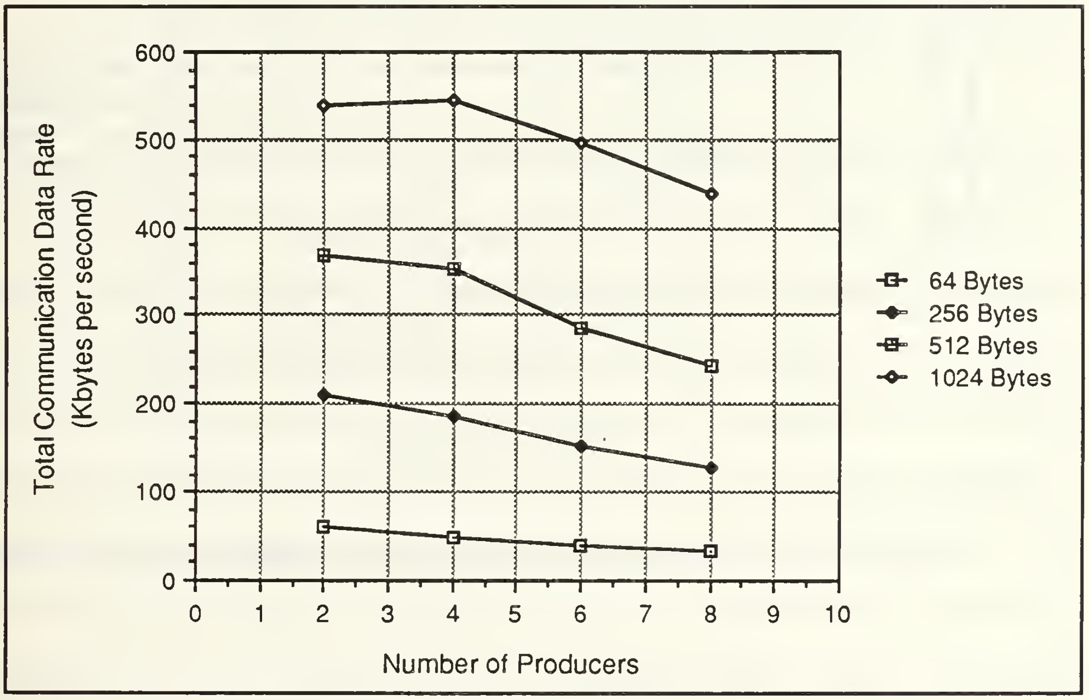


Figure 5.9. **Shared Hardware Link Communication Rates**

3. Multiple Link Effects on Communication Rate

Testing of the message-passing interface with a varying number of links active was performed for the message-passing interface. The results of this testing are shown in Figure 5.10. This testing revealed the same type of kernel data rate limitation found when testing the shared-memory model interface. Although the data rate of the message-passing kernel is significantly greater than that of the shared memory interface, the message passing kernel's data rate limit remains less than the capacity of one hardware communications link.

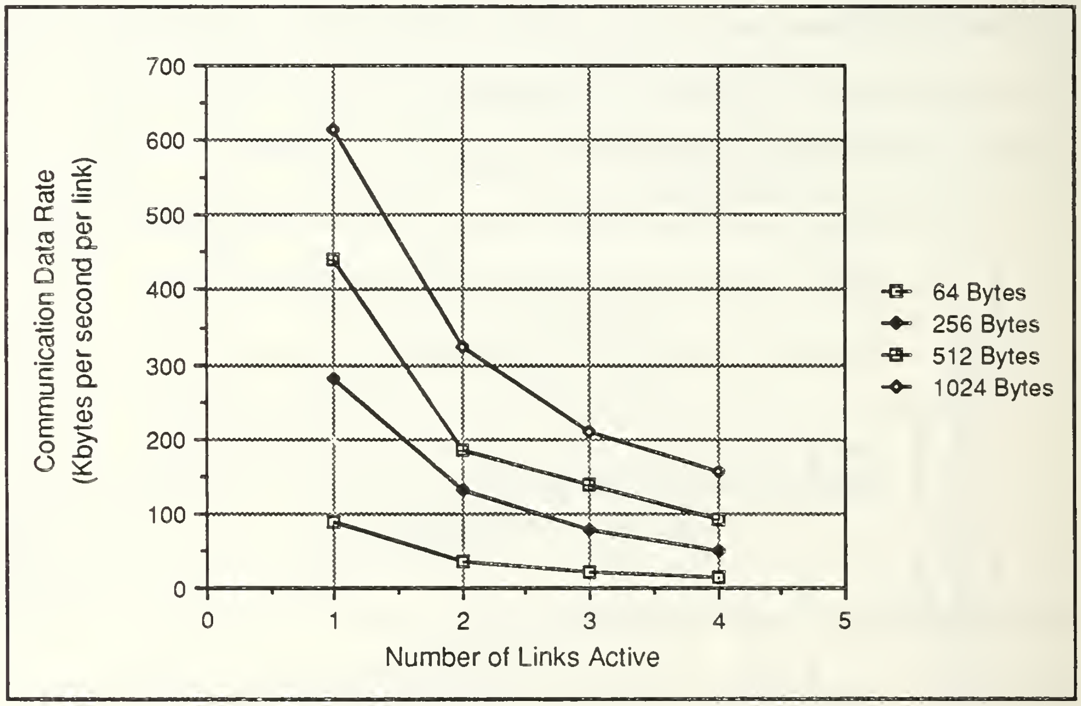


Figure 5.10. **Multiple Hardware Link Communication Rates**

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

This thesis has evaluated the performance of an isolated Transputer and the performance of two abstract programming interfaces with distributed kernels for a network of Transputers. The results of the evaluation show that, although the use of a distributed kernel reduces the effective performance of individual Transputers in the network, the performance remains relatively high. In general, the performance of the message-passing interface was superior to that of the shared-memory model interface. This comparison should not, however, be taken as absolute. There are improvements discussed in this chapter that can be made to both interfaces which could significantly affect this comparison.

Thus far, all evaluation of the abstract programming interfaces has been based on testing. The programming interfaces also need to be examined on the basis of whether or not they can be effectively used in the development of distributed programs. By its nature, such an assessment is subjective and prone to be influenced by one's prior experience and personal programming style preferences. Based on the experience of developing test and example programs for use with the interfaces, it appears that both of the interfaces do simplify the programming of a physical network of Transputers. The primary factor in this is that the programmer is isolated from having to consider a specific physical configuration of a network at all stages of program development. Only after the program is developed need a particular configuration be considered, at which point the physical configuration is described by a set of simple data tables.

Both interfaces accomplish the goal of isolating the software designer from physical configuration considerations. There was, however, no clear choice as to which interface methodology was the overall "best" from a programming standpoint. Each methodology had certain advantages and disadvantages. In general, the initial design, development, and coding of a distributed program was most effectively accomplished using the message-passing model. However, once past the initial implementation of a program, modifications, either to add additional functionality or to correct initial design errors, are usually required. In most cases, these modifications and additions were easier to make when using the shared-memory model interface.

B. RECOMMENDATIONS

As was mentioned previously, the programs for these prototype interface implementations are not fully optimized. As a direction for further work in developing the abstract interfaces for a network of Transputers, several modifications to the interface implementation should be considered. In general, the same types of changes can be made to both interfaces to improve performance. Some suggested changes are listed as follows:

- The message packet format can be improved. The current message format is actually scheduled as three separate transmissions: transmission of the header, transmission of a size value, and transmission of the data array. Efficiency can be improved by reducing the number of required transmissions. For example, the header information and the data array could be combined into a single array transmission preceded by a size transmission. This would eliminate one scheduled transmission each time a message is handled.

- The transmission of messages within the kernel uses OCCAM communications channels. This transmission is performed by processor-controlled memory-to-memory transfer. When the size of a data message is large, this decreases the performance of a node. A simple memory manager could be incorporated into the kernel to allocate space for messages in transit so that the transfer of messages within the kernel could be performed by communicating pointers or handles to the messages via the OCCAM channels.

- In the case of the shared-memory model, the interface only transfers data from a shared-memory segment to a remote node when an application program module at the remote node requests the data. If it is known in advance that certain remote nodes or modules will require frequent access to a node's shared memory segment, performance could be improved by having the distributed kernel automatically and periodically transfer selected segments of globally shared memory between nodes.

- Many of the data structures within the kernels are sparse. As the size of a network becomes larger, the storage required for these structures will also become larger. At some point, it is likely that a compressed storage scheme for the kernel data structures will need to be adopted.

In addition to these performance improvements, the kernel should be modified to include a distributed algorithm for examining the network to determine a network's physical configuration. Use of such an algorithm would save the software designer from having to specify the physical configuration of a network as the last step of the design process. More importantly, however, to support fault tolerance, such an algorithm could be used to identify an altered physical configuration so that alternate message routing paths could be established.

APPENDIX A

DETAILED TRANSPUTER TIMING TEST CONFIGURATION AND SOURCE CODE

A. SUMMARY

The purpose of the timing test configuration and associated test code is to provide a general framework for testing Transputer performance characteristics. The detailed configuration of the test set-up is shown in Figure A.1. The test set-up consists of a central "target" Transputer and four "satellite" Transputers, each attached to the target Transputer by a communications link. In addition, there are associated Transputers to perform the functions of control and of data routing and recording.

B. SOURCE CODE

1. Configuration Section

```
----------------------------------------------------------------------
----------------------------------------------------------------------
                        TIMING TEST PROGRAM
----------------------------------------------------------------------
----------------------------------------------------------------------

{{{  link definitions
VAL   link0out  IS  0 :
VAL   link1out  IS  1 :
VAL   link2out  IS  2 :
VAL   link3out  IS  3 :
VAL   link0in   IS  4 :
VAL   link1in   IS  5 :
VAL   link2in   IS  6 :
VAL   link3in   IS  7 :
}}}
{{{  declarations
CHAN OF ANY  arm.mid.0,   arm.mid.1,   arm.mid.2,   arm.mid.3,
             mid.arm.0,   mid.arm.1,   mid.arm.2,   mid.arm.3,
```

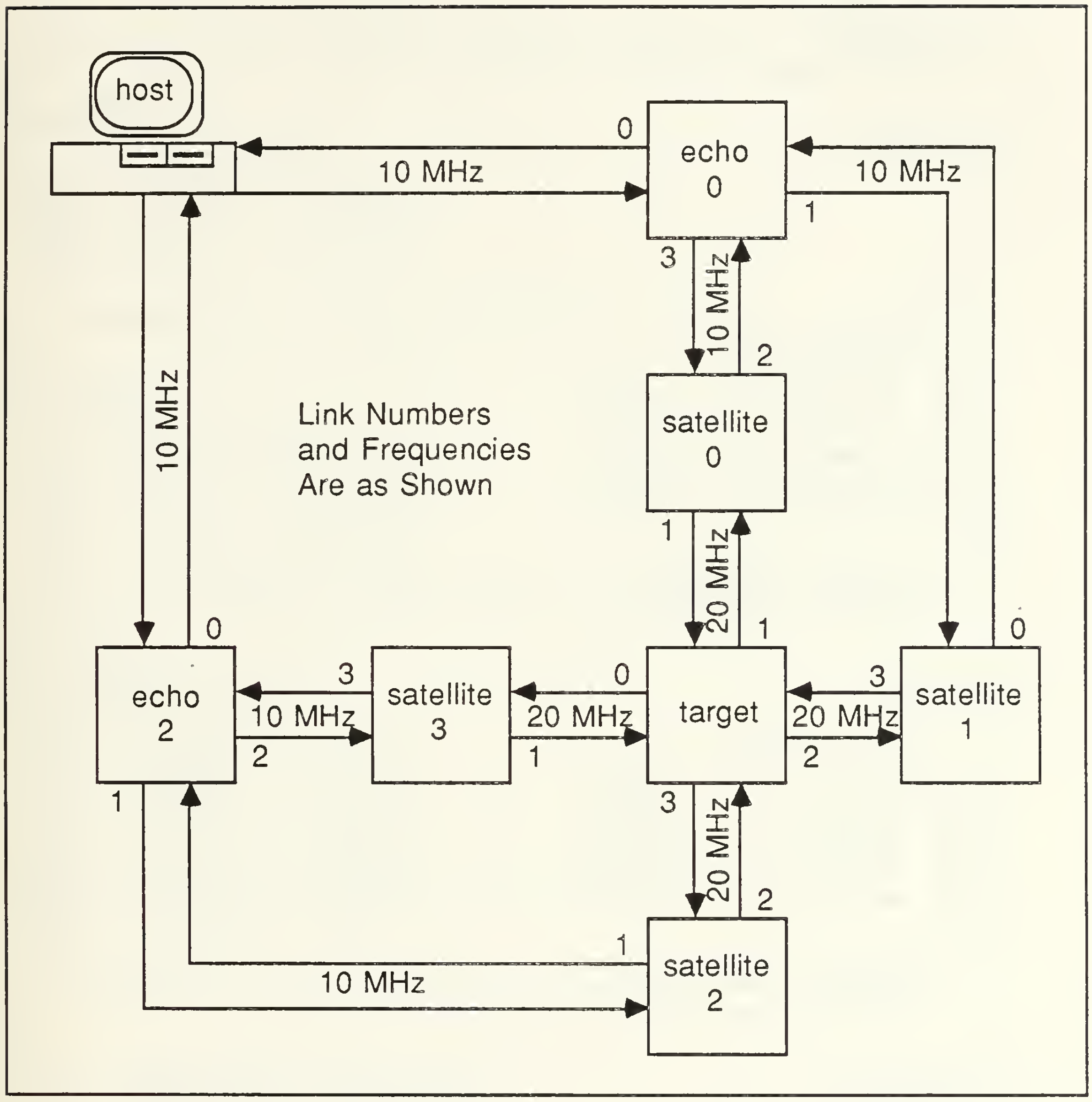


Figure A.1. **Detailed Test Configuration**

```
              arm.echo.0,  arm.echo.1,  arm.echo.2,  arm.echo.3,
              echo.arm.0,  echo.arm.1,  echo.arm.2,  echo.arm.3,
              host.echo.0, host.echo.2,
              echo.host.0, echo.host.2 :
}}}

{{{  SC target
...target procedure
}}}
{{{  SC satellite
...satellite procedure
}}}
{{{  SC echo
...echo procedure
}}}

PLACED PAR

  {{{  echo 0
  PROCESSOR 10 T4
    {{{  channel placements
    PLACE  echo.host.0  AT  link0out :
    PLACE  host.echo.0  AT  link0in  :
    PLACE  echo.arm.0   AT  link3out :
    PLACE  arm.echo.0   AT  link3in  :
    PLACE  echo.arm.1   AT  link1out :
    PLACE  arm.echo.1   AT  link1in  :
    }}}
    echo(echo.host.0, host.echo.0,
         echo.arm.0,  arm.echo.0,
         echo.arm.1,  arm.echo.1, 0)
  }}}
  {{{  echo 2
  PROCESSOR 11 T4
    {{{  channel placements
    PLACE  echo.host.2  AT  link0out :
    PLACE  host.echo.2  AT  link0in  :
    PLACE  echo.arm.2   AT  link1out :
    PLACE  arm.echo.2   AT  link1in  :
    PLACE  echo.arm.3   AT  link2out :
    PLACE  arm.echo.3   AT  link2in  :
    }}}
    echo(echo.host.2, host.echo.2,
         echo.arm.2,  arm.echo.2,
         echo.arm.3,  arm.echo.3, 2)
  }}}
  {{{  satellite 0
  PROCESSOR 13 T4
    {{{  channel placements
    PLACE  arm.echo.0  AT  link2out :
    PLACE  echo.arm.0  AT  link2in  :
    PLACE  arm.mid.0   AT  link1out :
    PLACE  mid.arm.0   AT  link1in  :
    }}}
    satellite(arm.echo.0, echo.arm.0, arm.mid.0, mid.arm.0, 0)
```

```
}}}
{{{  satellite 1
PROCESSOR 21 T4
  {{{  channel placements
  PLACE  arm.echo.1  AT  link0out :
  PLACE  echo.arm.1  AT  link0in  :
  PLACE  arm.mid.1   AT  link3out :
  PLACE  mid.arm.1   AT  link3in  :
  }}}
  satellite(arm.echo.1, echo.arm.1, arm.mid.1, mid.arm.1, 1)
}}}
{{{  satellite 2
PROCESSOR 23 T4
  {{{  channel placements
  PLACE  arm.echo.2  AT  link1out :
  PLACE  echo.arm.2  AT  link1in  :
  PLACE  arm.mid.2   AT  link2out :
  PLACE  mid.arm.2   AT  link2in  :
  }}}
  satellite(arm.echo.2, echo.arm.2, arm.mid.2, mid.arm.2, 2)
}}}
{{{  satellite 3
PROCESSOR 12 T4
  {{{  channel placements
  PLACE  arm.echo.3  AT  link3out :
  PLACE  echo.arm.3  AT  link3in  :
  PLACE  arm.mid.3   AT  link1out :
  PLACE  mid.arm.3   AT  link1in  :
  }}}
  satellite(arm.echo.3, echo.arm.3, arm.mid.3, mid.arm.3, 3)
}}}
{{{  target
PROCESSOR 20 T4
  {{{  channel placements
  PLACE  mid.arm.0  AT  link1out :
  PLACE  arm.mid.0  AT  link1in  :
  PLACE  mid.arm.1  AT  link2out :
  PLACE  arm.mid.1  AT  link2in  :
  PLACE  mid.arm.2  AT  link3out :
  PLACE  arm.mid.2  AT  link3in  :
  PLACE  mid.arm.3  AT  link0out :
  PLACE  arm.mid.3  AT  link0in  :
  }}}
  target(mid.arm.0, arm.mid.0, mid.arm.1, arm.mid.1,
         mid.arm.2, arm.mid.2, mid.arm.3, arm.mid.3)
}}}
```

2. <u>Target Node Procedure</u>

```
--------------------------------------------------------------------------
PROC target(CHAN OF ANY out.link.0, in.link.0,  out.link.1, in.link.1,
                        out.link.2, in.link.2,  out.link.3, in.link.3)
--------------------------------------------------------------------------
{{{  description
--  This procedure performs communications with adjacent nodes and performs
--  calculations in a loop to measure the interaction of communication and
--  calculation.  In general, a timer is started and the communications and
--  the calculation is started.  When the communications are complete, the
--  timer and the looping calculation are stopped.  The time for the
--  communication and the number of loops performed during this time is
--  extracted and reported.
}}}
--------------------------------------------------------------------------------

  {{{  declarations
  VAL     else        IS TRUE :
  VAL     one.second IS [1000000/64, 1000000] :

  VAL     ave.count  IS  1 :

  {{{  priority codes
  VAL     low         IS 0 :
  VAL     high        IS 1 :
  }}}

  {{{  speed codes
  VAL     slow        IS 0 :
  VAL     fast        IS 1 :
  }}}

  {{{  operation codes
  VAL     none        IS  0 :
  VAL     null        IS  1 :
  VAL     assign      IS  2 :
  VAL     add         IS  3 :
  VAL     sub         IS  4 :
  VAL     mult        IS  5 :
  VAL     div         IS  6 :
  VAL     move100     IS  7 :
  VAL     move1000    IS  8 :
  }}}

  {{{  timedata.tsr
  VAL          block.size    IS  100000:
  VAL          packet.counts IS [1,2,5,10,20,50,100,200,500,
                                  1000,2000,5000,10000,20000,50000,100000] :
  }}}

  CHAN OF BOOL status :
  CHAN OF INT  result :
  }}}

  {{{  test set

  VAL     loop.op    IS  null :
  VAL     op.count   IS  4 :
  VAL     loop.pri   IS  low :
  VAL     loop.loc   IS  fast :
  VAL     comm.pri   IS  high :
  VAL     comm.loc   IS  fast :
}}}

PRI PAR
  {{{  do communication and reporting
  {{{  process declarations
  INT     trigger :
  INT     packet.count, packet.size :
  INT     loop.count :
  INT     start.time, stop.time :

  [block.size]BYTE link.data:
  PLACE link.data    AT 4096:

  TIMER   clock :
  }}}
  SEQ in.links = 0 FOR 5
    SEQ out.links = 0 FOR 5
      SEQ packet.count.index = 0 FOR SIZE packet.counts
        SEQ
          {{{  initialize for test
          packet.count := packet.counts[packet.count.index]
          packet.size := block.size / packet.count
          }}}
          {{{  take care of triggering
          in.link.0 ? trigger
          PAR
            out.link.0 ! trigger
            out.link.1 ! trigger
            out.link.2 ! trigger
            out.link.3 ! trigger
          status     ! FALSE
          }}}
          {{{  start the timer
          clock ? start.time
          }}}
          {{{  do communication
          IF
            (in.links + out.links) = 0
              {{{  time one second delay for no links active case
              clock  ? AFTER start.time + one.second[comm.pri]
              }}}
            else
              {{{  set up links
              PAR
                {{{  input links
                {{{  start link 0 input
                IF
                  in.links > 0
                    SEQ i = 0 FOR packet.count
                      in.link.0 ? [link.data FROM 0 FOR packet.size]
                  else
                    SKIP
                }}}
                {{{  start link 1 input
```

```
IF
  in.links > 1
    SEQ i = 0 FOR packet.count
      in.link.1 ? [link.data FROM 0 FOR packet.size]
  else
    SKIP
}}}
{{{  start link 2 input
IF
  in.links > 2
    SEQ i = 0 FOR packet.count
      in.link.2 ? [link.data FROM 0 FOR packet.size]
  else
    SKIP
}}}
{{{  start link 3 input
IF
  in.links > 3
    SEQ i = 0 FOR packet.count
      in.link.3 ? [link.data FROM 0 FOR packet.size]
  else
    SKIP
}}}
}}}
{{{  ouput links
{{{  start link 0 output
IF
  out.links > 0
    SEQ i = 0 FOR packet.count
      out.link.0 ! [link.data FROM 0 FOR packet.size]
  else
    SKIP
}}}
{{{  start link 1 output
IF
  out.links > 1
    SEQ i = 0 FOR packet.count
      out.link.1 ! [link.data FROM 0 FOR packet.size]
  else
    SKIP
}}}
{{{  start link 2 output
IF
  out.links > 2
    SEQ i = 0 FOR packet.count
      out.link.2 ! [link.data FROM 0 FOR packet.size]
  else
    SKIP
}}}
{{{  start link 3 output
IF
  out.links > 3
    SEQ i = 0 FOR packet.count
      out.link.3 ! [link.data FROM 0 FOR packet.size]
  else
```

```
                    SKIP
                  }}}
                  }}}
                }}}
            }}}
            {{{  stop the loop count
            status ! TRUE
            }}}
            {{{  stop the timer
            clock ? stop.time
            }}}
            {{{  get loop count results
            result     ? loop.count
            }}}
            {{{  report results
            out.link.0 ! ave.count
            out.link.0 ! loop.op; op.count; loop.pri; loop.loc
            out.link.0 ! comm.pri; comm.loc; in.links; out.links
            out.link.0 ! block.size; packet.count; packet.size
            out.link.0 ! start.time; stop.time; loop.count
            }}}
}}}
{{{  do looping calculation
{{{  process declarations
BOOL    done :
INT     iteration.count :
INT     a,  b,  c :
[1000]BYTE move.data :
PLACE move.data AT 4096 :
}}}
SEQ
  {{{   init calc variables
  b := #FFFFFFE
  c := #FFF
  }}}
  SEQ in.links = 0 FOR 5
    SEQ out.links = 0 FOR 5
      SEQ packet.count.index = 0 FOR SIZE packet.counts
        {{{  do calculation loop
        SEQ
          iteration.count := 0
          status ? done
          WHILE NOT done
            PRI ALT
              status ? done
                SKIP
              TRUE & SKIP
                SEQ
                  iteration.count := iteration.count + 1
                  -- Calulation to be done in the loop goes here
          result ! iteration.count
        }}}
}}}
{{{  COMMENT do no looping calculation
{{{  do no looping calculation
```

```
    {{{  process declarations
    BOOL    done :
    }}}
    SEQ in.links = 0 FOR 5
      SEQ out.links = 0 FOR 5
        SEQ packet.count.index = 0 FOR SIZE packet.counts
          {{{  do wait for synchronization with communication process
          SEQ
            status ? done
            status ? done
            result ! 0
          }}}
    }}}
    }}}

:
------------------------------------------------------------------------------
```

3. Satellite Node Procedure

```
------------------------------------------------------------------------------
PROC satellite(CHAN OF ANY to.echo, from.echo, to.mid, from.mid,
               VAL INT my.tag)
------------------------------------------------------------------------------
{{{  description
--  This procedure provides for sourcing and sinking communications to place
--  a communications load on the target node of the test configuration.  If the
--  procedure is placed in a the data reporting path from the target node
--  to the host system, the satellite also passes along the data from the
--  target node.  Packet sizes transmitted from this node are the same size
--  as the packets being received by the target.
}}}
------------------------------------------------------------------------------

  {{{  declarations
  VAL     else       IS TRUE :

  INT     trigger :
  INT     packet.count, packet.size :
  INT     start.time, stop.time :

  INT     ave.count :
  INT     loop.op, op.count, loop.pri, loop.loc :
  INT     comm.pri, comm.loc, mid.in, mid.out :
  INT     mid.block, mid.packet.count, mid.packet.size :
  INT     mid.start, mid.stop, mid.loop :

  {{{  timedata.tsr
  VAL          block.size    IS  100000:
  VAL          packet.counts IS [1,2,5,10,20,50,100,200,500,
                                 1000,2000,5000,10000,20000,50000,100000] :
  }}}

  [block.size]BYTE link.data:
  PLACE   link.data   AT 4096 :
```

```
TIMER   clock :
}}}

PRI PAR
  SEQ in.links = 0 FOR 5
    SEQ out.links = 0 FOR 5
      SEQ packet.count.index = 0 FOR SIZE packet.counts
        SEQ
          {{{  if 'primary' satellite pass on the trigger
          IF
            my.tag = 0
              SEQ
                from.echo  ? trigger
                to.mid     ! trigger
            else
              SKIP
          }}}
          {{{  wait for target start trigger
          from.mid ? trigger
          }}}
          {{{  initialize for test set
          packet.count  := packet.counts[packet.count.index]
          packet.size  := block.size / packet.count
          clock ? start.time
          }}}
          {{{  set up links
          PAR
            {{{  start link input
            IF
              out.links > my.tag
                SEQ i = 0 FOR packet.count
                  from.mid ? [link.data FROM 0 FOR packet.size]
              else
                SKIP
            }}}
            {{{  start link output
            IF
              in.links > my.tag
                SEQ i = 0 FOR packet.count
                  to.mid ! [link.data FROM 0 FOR packet.size]
              else
                SKIP
            }}}
          }}}
          {{{  complete test set
          clock ? stop.time
          }}}
          {{{  if 'primary' satellite pass on the mid results
          IF
            my.tag = 0
              SEQ
                from.mid ? ave.count
                from.mid ? loop.op; op.count; loop.pri; loop.loc
                from.mid ? comm.pri; comm.loc; mid.in; mid.out
```

```
                  from.mid ? mid.block; mid.packet.count; mid.packet.size
                  from.mid ? mid.start; mid.stop; mid.loop
                  to.echo  ! ave.count
                  to.echo  ! loop.op; op.count; loop.pri; loop.loc
                  to.echo  ! comm.pri; comm.loc; mid.in; mid.out
                  to.echo  ! mid.block; mid.packet.count; mid.packet.size
                  to.echo  ! mid.start; mid.stop; mid.loop
              else
                SKIP
            }}}
            {{{  report own timing results
            to.echo ! start.time; stop.time
            }}}
    SKIP

:
------------------------------------------------------------------------------
```

4. Echoing (Data Routing) Node Procedure

```
------------------------------------------------------------------------------
PROC echo(CHAN OF ANY to.root, from.root,
                      to.arm.0, from.arm.0, to.arm.1, from.arm.1,
                      VAL INT my.tag)
------------------------------------------------------------------------------
{{{  description
--  This procedure echos timing results from the target and satellite nodes to
--  the host system.  The echoing procedure placed in the routing path from the
--  target to the host echos all the host data.
}}}
------------------------------------------------------------------------------

  {{{  declarations
  VAL    else       IS TRUE :

  {{{  timedata.tsr
  VAL          block.size    IS  100000:
  VAL          packet.counts IS [1,2,5,10,20,50,100,200,500,
                                 1000,2000,5000,10000,20000,50000,100000] :
  }}}

  INT     ave.count :
  INT     loop.op, op.count, loop.pri, loop.loc :
  INT     comm.pri, comm.loc, mid.in, mid.out :
  INT     mid.block, mid.packet.count, mid.packet.size :
  INT     mid.start, mid.stop, mid.loop :

  INT     trigger :
  INT     arm.start, arm.stop :

  }}}

  SEQ in.links = 0 FOR 5
    SEQ out.links = 0 FOR 5
      SEQ packet.count.index = 0 FOR SIZE packet.counts
```

```
        SEQ
          {{{  if 'primary' pass on the start trigger
          IF
            my.tag = 0
              SEQ
                from.root    ? trigger
                to.arm.0     ! trigger
            else
              SKIP
          }}}
          {{{  if 'primary' echo pass on the target results
          IF
            my.tag = 0
              SEQ
                from.arm.0 ? ave.count
                from.arm.0 ? loop.op; op.count; loop.pri; loop.loc
                from.arm.0 ? comm.pri; comm.loc; mid.in; mid.out
                from.arm.0 ? mid.block; mid.packet.count; mid.packet.size
                from.arm.0 ? mid.start; mid.stop; mid.loop
                to.root    ! ave.count
                to.root    ! loop.op; op.count; loop.pri; loop.loc
                to.root    ! comm.pri; comm.loc; mid.in; mid.out
                to.root    ! mid.block; mid.packet.count; mid.packet.size
                to.root    ! mid.start; mid.stop; mid.loop
            else
              SKIP
          }}}
          {{{  report satellite timing results
          from.arm.0 ? arm.start; arm.stop
          to.root    ! arm.start; arm.stop
          from.arm.1 ? arm.start; arm.stop
          to.root    ! arm.start; arm.stop
          }}}

:
--------------------------------------------------------------------------
```

5. Host (Data Recording) Procedure

```
--------------------------------------------------------------------------
PROC   root(CHAN OF ANY keyboard, screen,
         [4]CHAN OF ANY from.u.filer, to.u.filer,
            CHAN OF ANY from.fold, to.fold,
                        from.filer, to.filer)
--------------------------------------------------------------------------
{{{  description
--  This procedure runs on the host system.  It triggers the start of a test
--  run and gathers data from the network upon completion of the test run.
--  The data gathered is sent to a disk file on the host system for later
--  evaluation or transfer.  A subset of the gathered data is displayed on
--  on the host's screen.
}}}
--------------------------------------------------------------------------

  {{{  TDS library references
```

```
#USE "\tdsiolib\userio.tsr":
#USE "\tdsiolib\interf.tsr":
}}}

{{{  declarations
VAL         else          IS  TRUE :
VAL         trigger       IS  1 :
VAL         uS.per.tic    IS  [64, 1] :
VAL         tab           IS  BYTE 9 :

INT         id.length,result:
[63]BYTE    id.string:
CHAN OF ANY data.file:

{{{  timedata.tsr
VAL         block.size     IS  100000:
VAL         packet.counts IS [1,2,5,10,20,50,100,200,500,
                              1000,2000,5000,10000,20000,50000,100000] :
}}}

{{{  network report
INT         ave.count :
INT         loop.op, op.count, loop.pri, loop.loc :
INT         comm.pri, comm.loc, mid.in, mid.out :
INT         mid.block, mid.packet.count, mid.packet.size :
INT         mid.start, mid.stop, mid.loop, mid.time :
INT         arm.0.start, arm.0.stop, arm.0.time :
INT         arm.1.start, arm.1.stop, arm.1.time :
INT         arm.2.start, arm.2.stop, arm.2.time :
INT         arm.3.start, arm.3.stop, arm.3.time :
}}}

{{{  network channels
VAL         link0out      IS  0 :
VAL         link1out      IS  1 :
VAL         link2out      IS  2 :
VAL         link3out      IS  3 :
VAL         link0in       IS  4 :
VAL         link1in       IS  5 :
VAL         link2in       IS  6 :
VAL         link3in       IS  7 :

CHAN OF ANY from.net.0, from.net.2,
            to.net.0,   to.net.2 :

PLACE       to.net.0      AT  link2out :
PLACE       to.net.2      AT  link3out :
PLACE       from.net.0    AT  link2in  :
PLACE       from.net.2    AT  link3in  :
}}}

}}}

PAR
  {{{  set the filename and run the host system filer
```

```
SEQ
  [id.string FROM 0 FOR 6] := "time01"
  id.length := 6
  scrstream.to.server(data.file,
                      from.filer, to.filer,
                      id.length, id.string,
                      result)
}}}
{{{  run the test network
SEQ
  SEQ in.links = 0 FOR 5
    SEQ out.links = 0 FOR 5
      SEQ packet.count.index = 0 FOR SIZE packet.counts
        SEQ
          {{{  trigger network
          to.net.0 ! trigger
          }}}
          {{{  receive network results
          from.net.0 ? ave.count
          from.net.0 ? loop.op; op.count; loop.pri; loop.loc
          from.net.0 ? comm.pri; comm.loc; mid.in; mid.out
          from.net.0 ? mid.block; mid.packet.count; mid.packet.size
          from.net.0 ? mid.start; mid.stop; mid.loop

          from.net.0 ? arm.0.start; arm.0.stop
          from.net.0 ? arm.1.start; arm.1.stop
          from.net.2 ? arm.2.start; arm.2.stop
          from.net.2 ? arm.3.start; arm.3.stop

          mid.time := (mid.stop-mid.start)*uS.per.tic[comm.pri]

          arm.0.time := arm.0.stop-arm.0.start
          arm.1.time := arm.1.stop-arm.1.start
          arm.2.time := arm.2.stop-arm.2.start
          arm.3.time := arm.3.stop-arm.3.start
          }}}
          {{{  put results to file
          {{{  processor type
          write.int(data.file,800,1)
          write.char(data.file,tab)
          }}}
          {{{  processor speed
          write.int(data.file,20,1)
          write.char(data.file,tab)
          }}}
          {{{  link speed
          write.int(data.file,20,1)
          write.char(data.file,tab)
          }}}
          {{{  loop conditions
          write.int(data.file,loop.op,1)
          write.char(data.file,tab)
          write.int(data.file,op.count,1)
          write.char(data.file,tab)
          write.int(data.file,loop.pri,1)
          write.char(data.file,tab)
          write.int(data.file,loop.loc,1)
          write.char(data.file,tab)
          }}}
          {{{  link conditions
          write.int(data.file,comm.pri,1)
          write.char(data.file,tab)
          write.int(data.file,comm.loc,1)
          write.char(data.file,tab)
          write.int(data.file,mid.in,1)
          write.char(data.file,tab)
          write.int(data.file,mid.out,1)
          write.char(data.file,tab)
          }}}
          {{{  comm.conditions
          write.int(data.file,mid.block,1)
          write.char(data.file,tab)
          write.int(data.file,mid.packet.count,1)
          write.char(data.file,tab)
          write.int(data.file,mid.packet.size,1)
          write.char(data.file,tab)
          }}}
          {{{  target results
          write.int(data.file,mid.loop,1)
          write.char(data.file,tab)
          write.int(data.file,mid.time,1)
          write.char(data.file,tab)
          }}}
          {{{  satellite results
          write.int(data.file,arm.0.time,1)
          write.char(data.file,tab)
          write.int(data.file,arm.1.time,1)
          write.char(data.file,tab)
          write.int(data.file,arm.2.time,1)
          write.char(data.file,tab)
          write.int(data.file,arm.3.time,1)
          newline(data.file)
          }}}
          }}}
          {{{  put results to screen
          write.int(screen, in.links,3)
          write.int(screen, out.links,3)
          write.int(screen, mid.packet.count,7)
          write.int(screen, mid.packet.size,7)
          write.int(screen, mid.loop, 9)
          write.int(screen, mid.time, 9)
          write.int(screen, arm.0.time, 9)
          write.int(screen, arm.1.time, 9)
          write.int(screen, arm.2.time, 9)
          write.int(screen, arm.3.time, 9)
          newline(screen)
          }}}
      {{{  terminate host system filer
      data.file ! 24
      }}}
```

```
    }}}

  :
-------------------------------------------------------------------------------
```

APPENDIX B

TIMING DATABASE DESCRIPTION

A. DISCUSSION

The large volume of data generated by individual timing tests was loaded into a database management system to facilitate ease of access and aggregation of any future test results. The Ingres Relational Database Management System, currently available on the departmental mini-computer, was selected for this purpose. The Ingres database management system provides a wide range of tools for accessing and manipulating data. These tools range from a database query language that can be used to program complex data manipulations and operations to a completely menu-driven interactive shell for accessing the database.

This appendix documents the format of the Ingres timing database. Also, some examples are provided on ways in which timing data can be accessed through the use of the query language. Complete documentation for the use of the many features of the Ingres relational database system may be found in [Re85].

B. DATABASE DESCRIPTION

A relational database consists of a set of relations. A relation is, in turn, composed of a set of data fields. A relation can be thought of as a table where the data fields represent the columns of the table. One row of the table then represents one individual set of data entered in the table.

In the timing database, there are several relations that are defined. The first of these relations, shown in Table B.1, is the relation that actually holds the timing test data. This table shows each of the fields in the timing relation, the type of data the field is formatted for and the length of the data field (in bytes).

TABLE B.1

TIMING DATA RELATION DEFINITION

Relation Name: timing

column name	type	length
processor	integer	2
procspeed	integer	1
linkspeed	integer	1
opcode	integer	1
opcount	integer	1
looppri	integer	1
looploc	integer	1
commpri	integer	1
commloc	integer	1
linksin	integer	1
linksout	integer	1
blocksize	integer	4
packcount	integer	4
packsize	integer	4
loopcount	integer	4
looptime	integer	4
arm0time	integer	4
arm1time	integer	4
arm2time	integer	4
arm3time	integer	4

The name of this relation is “timing.” Note that the fields defined in this relation are the same as the data elements written to the host system file by the timing test program of Appendix A.

There are two additional relations in the timing test database. These relations, shown in Tables B.2, B.3, and B.4, provide for translating the numerically encoded calculation loop operation codes, the timing test priorities, and memory locations in the “timing” relation into a text description of the corresponding test condition.

TABLE B.2

RELATION DEFINITION FOR CONVERTING ENCODED OPERATION CODE NAMES

Relation Name: opcodenames

column name	type	length
opcode	integer	1
opcodename	character	32

TABLE B.3

RELATION DEFINITION FOR CONVERTING ENCODED PRIORITY NAMES

Relation Name: prinames

column name	type	length
pri	integer	1
priname	character	32

TABLE B.4

RELATION DEFINITION FOR CONVERTING

ENCODED LOCATION NAMES

Relation Name: locnames

column name	type	length
loc	integer	1
locname	character	32

C. EXAMPLES

There are three database operations that are provided as examples of query language interaction with the database. These operations are: loading an external text file of data into the database, retrieving information from the database, and, finally, transferring data from the database to an external text file.

1. Data Loading

Figure B.1 is a listing of the query language instructions for transferring a file of test data from the timing test program into the timing database. For the most part, these instructions are self explanatory. Of note are the formatting codes, "c0tab" or "c0nl," associated with each of the fields. These formatting codes mean that the text file representation of of a field's data is a variable length character string terminated by a tab character or by a new line character.

```
copy timing
    (
    processor = c0tab,
    procspeed = c0tab,
    linkspeed = c0tab,
    opcode    = c0tab,
    opcount   = c0tab,
    looppri   = c0tab,
    looploc   = c0tab,
    commpri   = c0tab,
    commloc   = c0tab,
    linksin   = c0tab,
    linksout  = c0tab,
    blocksize = c0tab,
    packcount = c0tab,
    packsize  = c0tab,
    loopcount = c0tab,
    looptime  = c0tab,
    arm0time  = c0tab,
    arm1time  = c0tab,
    arm2time  = c0tab,
    arm3time  = c0nl
    )
from "/work/ingres/time.file"
```

Figure B.1. **Query Language Listing for Loading the Database**

2. Data Retrieval

Figures B.2 and B.3 are examples of query language data retrievals. In Figure B.2, the time (in µseconds) for single link bidirectional communication of 100,000 bytes with a packet size of 10 bytes is retrieved for all types of processors. The results of this retrieval are also displayed.

In Figure B.3, the same retrieval is processed but, instead of displaying the results, the results of the query are used to form the new relation "new_relation."

```
retrieve
    (
    timing.processor,
    timing.looptime
    )
  where
    timing.linksin          = 1                   and
    timing.linksout         = 1                   and
    timing.packsize         = 10                  and
    timing.opcode           = opcodenames.opcode  and
    opcodenames.opcodename  = "null loop"
  sort by
    timing.processor,
    timing.looptime


Executing . . .


|proces|looptime     |
|--------------------|
|   414|       191659|
|   800|       123143|
|--------------------|
```

Figure B.2. **Example of Retrieval for Display**

```
retrieve into new_relation
    (
    timing.processor,
    timing.looptime
    )
  where
    timing.linksin          = 1                   and
    timing.linksout         = 1                   and
    timing.packsize         = 10                  and
    timing.opcode           = opcodenames.opcode  and
    opcodenames.opcodename  = "null loop"
  sort by
    timing.processor,
    timing.looptime
```

Figure B.3. **Example of Retrieval for Forming a New Relation**

As can be seen in Table B.5, which lists the characteristics for this new relation, the fields in the new relation inherit their characteristics from the relation from which the field originated.

TABLE B.5

RELATION DEFINITION FOR THE RETRIEVED NEW_RELATION

Relation Name: new_relation

column name	type	length
processor	integer	2
looptime	integer	4

3. Data UnLoading

Figure B.4 shows the query language listing for transferring the previously created relation "new_relation" to an external text file. As with the loading of data into the database, a formatting code is associated with each field to be transferred to the external text file. The interpretation of these formatting codes is the same as for the formatting codes for loading the database.

```
copy new_relation
    (
    processor   = c0tab,
    looptime    = c0nl
    )
  into "/work/ingres/text.file"
```

Figure B.4. **Query Language Listing for Unloading the Database**

APPENDIX C

DETAILED SOURCE CODE FOR THE SHARED MEMORY ABSTRACT INTERFACE LIBRARY

A. SYMBOLIC CONSTANTS

```
{{{  LIB  this is LIBRARY "\ecslib\ecssymb.tsr"
...  Library ID
{{{  VAL Declarations
{{{  useful symbolic constants
VAL    ELSE                  IS  TRUE :
VAL    NIL                   IS  -1 :
VAL    OUTPUT                IS   0 :
VAL    INPUT                 IS   1 :
}}}
{{{  definition of system limits
--  define the maximum size of an individual message
VAL    MAX.MESSAGE.SIZE        IS  1028 :

--  define the maximum size of one node's part of the
--  globally shared memory
VAL    MAX.DATA.PER.NODE       IS  4096 :

--  define the maximum number of event counts allowed
--  in the system
VAL    MAX.COUNTS              IS  256  :

--  define the maximum number of event counts to be
--  maintained by one node
VAL    MAX.COUNTS.PER.NODE     IS  16 :

--  define the maximum space allocation for the waiting
--  process lists
VAL    MAX.AWAIT.QUE.ENTRIES   IS  16 :

--  define the maximum number of processes per node
VAL    MAX.PROC.PER.NODE       IS  16 :
}}}
{{{  action codes for communications packets
VAL    READ.CMD              IS  10 :
VAL    READ.REP              IS  11 :

VAL    AWAIT.CMD             IS  20 :
VAL    AWAIT.REP             IS  21 :
VAL    AWAIT.QUE.FULL        IS  22 :

VAL    ADVANCE.CMD           IS  30 :
VAL    ADVANCE.REP           IS  31 :
```

```
VAL    TICKET.CMD          IS  40 :
VAL    TICKET.REP          IS  41 :

VAL    GET.CMD             IS  50 :
VAL    GET.REP             IS  51 :

VAL    PUT.CMD             IS  60 :
VAL    PUT.REP             IS  61 :
}}}
}}}
}}}
```

B. KERNEL PROCEDURE

```
------------------------------------------------------------------------------
PROC ecs.kernel([][2]CHAN OF ANY links,
                VAL INT         node.id,
                VAL []INT       count.node,
                VAL []INT       count.size,
                VAL []INT       node.link)
------------------------------------------------------------------------------
{{{ description
-- Event Counts and Sequencers for the Transputer

-- This program is an distributed kernel for implementing event counts and
-- sequencers on a network of transputers.  It should be run at high
-- priority in parallel with application program modules.  The application
-- program modules are linked to the kernel via bidirectional communication
-- channels that are elements of the links channel array parameter.
}}}
------------------------------------------------------------------------------

  {{{ libraries
  #USE "\ecslib\ecssymb.tsr"
  }}}
  {{{ local declarations
  {{{ local abbreviations
  VAL INT num.nodes      IS       SIZE node.link :
  VAL INT num.counts     IS       SIZE count.node :
  VAL INT no.links       IS       SIZE links :
  VAL INT no.h.links     IS       4 :
  VAL INT no.s.links     IS       no.links - no.h.links :
  }}}
  {{{ channel declarations and assignments
  -- channels for kernel control of the hard/physical communications links
  [2*no.h.links]CHAN OF ANY      hard.links :
  PLACE   hard.links     AT       0 :
  }}}
  {{{ communications data structures
  -- basic format of a received communications header
  [6]INT  header :
  []INT   short.header   IS       [header FROM 0 FOR 2] :
  INT     action.code    IS       header[0] :
  INT     count.id       IS       header[1] :
```

```
INT     to.node       IS      header[2] :
INT     to.proc       IS      header[3] :
INT     from.node     IS      header[4] :
INT     from.proc     IS      header[5] :

INT     data.size :
-- potential formats of received data arrays
[MAX.MESSAGE.SIZE]BYTE  data.array :
INT     count.value   RETYPES  [data.array FROM 0 FOR 4] :
INT     index         RETYPES  [data.array FROM 0 FOR 4] :
INT     seg.size      RETYPES  [data.array FROM 4 FOR 4] :
}}}
{{{  event count data structures
-- variables to track allocation of shared resources
INT     node.counts,
        base.count,
        next.free :

-- storage allocation for kernel data structures
[MAX.COUNTS]INT                 count.array.indices :
[MAX.COUNTS.PER.NODE][4]INT     count.array :
[MAX.DATA.PER.NODE]BYTE         node.data :
[MAX.AWAIT.QUE.ENTRIES][4]INT   node.awaits :
[MAX.AWAIT.QUE.ENTRIES]INT      free.list :
}}}
}}}
{{{  procedures
{{{  PROC buffer.in.link(chan.in,chan.out)
-------------------------------------------------------------------------------
PROC buffer.in.link(CHAN OF ANY in.link, out.link)
-------------------------------------------------------------------------------
{{{  description
-- This procedure provides a 'soft' buffer for hard link input and output.
-- This buffer increases the overall throughput of the node.
}}}
-------------------------------------------------------------------------------

  {{{  declarations
  [MAX.MESSAGE.SIZE]BYTE         data.array.0,
                                 data.array.1 :
  [6]INT                         header.0,
                                 header.1 :
  INT                            data.size.0,
                                 data.size.1 :
  }}}

  SEQ
    in.link   ?  header.0; data.size.0::data.array.0
    WHILE TRUE
      SEQ
        PAR
          out.link  !  header.0; data.size.0::data.array.0
          in.link   ?  header.1; data.size.1::data.array.1
        PAR
          out.link  !  header.1; data.size.1::data.array.1
        in.link    ?  header.0; data.size.0::data.array.0

:
--------------------------------------------------------------------------------
}}}
{{{  PROC buffer.out.link(chan.in,chan.out)
--------------------------------------------------------------------------------
PROC buffer.out.link(CHAN OF ANY in.link, out.link)
--------------------------------------------------------------------------------
{{{  description
--  This procedure buffers output to hardware communications links.
--  To improve performance, the buffer is sized to hold one message
--  for each process at the node.  The buffer is organized in a ring
--  configuration.  WARNING:  This implementation uses variables
--  shared between parallel processes.  The ring buffer itself, and
--  next in and out pointers are shared.  When this procedure is run
--  at high priority, the sequencing of the code guarantees that
--  there will be no access conflicts to these shared structures.
--  The purpose for this sharing is that the implementation is
--  slightly faster under average case loading conditions and is no
--  worse than a 'normal' request-next-item buffer under any conditions.
}}}
--------------------------------------------------------------------------------

  {{{  declarations
  -- a local alternate name for the system li,it used to
  -- size the buffer
  VAL INT buff.size     IS       MAX.PROC.PER.NODE :

  -- pointers to positions in the ring buffer
  INT                   next.in,
                        next.out :

  -- define the ring buffer
  [buff.size][6]INT     header :
  [buff.size]INT        data.size :
  [buff.size][MAX.MESSAGE.SIZE]BYTE   data.array :

  -- channels for communicating between the input and output
  -- parts of the buffer
  CHAN OF ANY           wake.in,
                        wake.out :
  }}}

  SEQ
    {{{  initialization
    next.in   := 0
    next.out  := 0
    }}}
    PAR
      {{{  buffer input
      {{{  local declarations
      INT       buff.no,
                any :
      }}}
      WHILE TRUE
        SEQ
          buff.no  := next.in\buff.size
          PRI ALT
            -- if the buffer is not full
            (next.in < (next.out + buff.size)) & SKIP
              SEQ
                -- input an item to the buffer
                in.link  ?  header[buff.no]; data.size[buff.no]::data.array[buff.no]
                next.in := next.in + 1
                -- if the buffer was empty, let the sleeping output know
                IF
                  (next.in - next.out) = 1
                    wake.out ! NIL
                  ELSE
                    SKIP
            -- the buffer is full, go to sleep until item output
            wake.in  ?  any
              SKIP
      }}}
      {{{  do output
      {{{  local declarations
      INT       buff.no,
                any :
      }}}
      WHILE TRUE
        SEQ
          buff.no  := next.out\buff.size
          PRI ALT
            -- if the buffer is not empty
            (next.in > next.out) & SKIP
              SEQ
                -- output a buffered item
                out.link ! header[buff.no]; data.size[buff.no]::data.array[buff.no]
                next.out := next.out + 1
                -- if the buffer was full, wake the sleeping input process
                IF
                  (next.in - next.out) = (buff.size - 1)
                    wake.in  !  NIL
                  ELSE
                    SKIP
            -- the buffer is empty, wait for a wake-up after some input
            wake.out  ?  any
              SKIP
      }}}

:
------------------------------------------------------------------------------
}}}
{{{  PROC send.packet(header, data.size, data.array)
------------------------------------------------------------------------------
PROC send.packet(VAL [6]INT  header, VAL INT data.size, []BYTE data.array)
------------------------------------------------------------------------------
{{{  description
-- This procedure packages messages for sending either to a remote node or
-- to a local process.
}}}
--------------------------------------------------------------------------

  {{{  declarations
  -- define subset of overall header for local communication
  VAL []INT   short.header   IS      [header FROM 0 FOR 2] :
  VAL INT     to.node        IS      header[2] :
  VAL INT     to.proc        IS      header[3] :
  }}}

  IF
    {{{  packet is to local procedure  --  return it locally
    to.node = node.id
      links[to.proc][OUTPUT] ! short.header; data.size::data.array
    }}}
    {{{  else packet is for remote node  --  pass it on
    ELSE
      links[node.link[to.node]][OUTPUT] ! header; data.size::data.array
    }}}

:
--------------------------------------------------------------------------
}}}
{{{  PROC process.packet(link.no)
--------------------------------------------------------------------------
PROC process.packet(VAL INT link.no)
--------------------------------------------------------------------------
{{{  description
-- This procedure performs a function based on the value of the action.code
-- element in the header of a communications packet.  These functions
-- correspond to the basic calls provided by the event counts and sequencers
-- procedures (read, advance, await, ticket, put and get).
}}}
--------------------------------------------------------------------------

  IF
    {{{  command packet requires processing by my node  --  handle it
    (to.node = node.id) AND (to.proc = 0)
      {{{  get characteristics of count.id
      INT  count.index     IS  count.array.indices[count.id] :
      INT  current.count   IS  count.array[count.index][0]  :
      INT  current.ticket  IS  count.array[count.index][1]  :
      INT  base            IS  count.array[count.index][2]  :
      INT  head            IS  count.array[count.index][3]  :
      }}}
      IF
        {{{  read command
        action.code = READ.CMD
          -- represent the count as an array of bytes
          []BYTE out.count RETYPES current.count :
          -- return the value of the count
          send.packet([READ.REP, count.id, from.node, from.proc, node.id, 0],
                      SIZE out.count, out.count)
        }}}
        {{{  advance command
        action.code = ADVANCE.CMD
          {{{  declarations

          -- Each time an advance is performed, the waiting process list
          -- associated with the target event count is checked to determine
          -- if the advanced count is a waited-for count.  If so, a wake-up
          -- message is sent to the suspended process.

          BOOL  more.to.wake:
          }}}
          SEQ
            {{{  initialization
            -- advance the event count
            current.count := current.count + 1
            more.to.wake := TRUE
            }}}
            WHILE more.to.wake
              IF
                {{{  no items on the waiting process list
                head = NIL
                  more.to.wake := FALSE
                }}}
                {{{  check the first item on the waiting process list
                ELSE
                  {{{  abbreviations
                  -- extract the wating process list entries for the first item
                  -- on the waiting process list.
                  INT   wait.count   IS  node.awaits[head][0] :
                  INT   wait.node    IS  node.awaits[head][1] :
                  INT   wait.proc    IS  node.awaits[head][2] :
                  INT   wait.next    IS  node.awaits[head][3] :
                  }}}
                  IF
                    {{{  count reached - wake the first process on the list
                    wait.count <= current.count
                      SEQ
                        {{{  send wakeup message
                        send.packet([AWAIT.REP, count.id, wait.node, wait.proc,
node.id, 0],
                                     0, data.array)
                        }}}
                        {{{  remove the first item, return array position to free
list

                        next.free := next.free - 1
                        free.list[next.free] := head
                        head := wait.next
                        }}}
                    }}}
                    {{{  count not reached - no more to wake
                    ELSE
                      more.to.wake := FALSE
                    }}}
                }}}
        }}}
```

```
          {{{  await command
          action.code = AWAIT.CMD
            IF
              {{{  wakeup if count already reached
              count.value <= current.count
                send.packet([AWAIT.REP, count.id, from.node, from.proc, node.id, 0],
                             0, data.array)
              }}}
              {{{  add to waiting process list if a future count
              ELSE
                IF
                  {{{  no room for an await list entry - send back await fail
                  next.free = MAX.AWAIT.QUE.ENTRIES
                    send.packet([AWAIT.QUE.FULL, count.id, from.node, from.proc,
node.id, 0],
                                   0, data.array)
                  }}}
                  {{{  is room for an await list entry - add the entry to the await
list
                  ELSE
                    {{{  abbreviations
                    -- extract and abbreviate the await list entries for the
                    -- next available position (from free list)
                    INT   wait.count  IS  node.awaits[free.list[next.free]][0] :
                    INT   wait.node   IS  node.awaits[free.list[next.free]][1] :
                    INT   wait.proc   IS  node.awaits[free.list[next.free]][2] :
                    INT   wait.next   IS  node.awaits[free.list[next.free]][3] :
                    }}}
                    SEQ
                      IF
                        {{{  handle list empty case
                        head = NIL
                          SEQ
                            wait.next := NIL
                            head := free.list[next.free]
                        }}}
                        {{{  handle insert at head of list case
                        count.value <= node.awaits[head][0]
                          SEQ
                            wait.next := head
                            head := free.list[next.free]
                        }}}
                        {{{  handle all other cases
                        ELSE
                          {{{  declarations
                          --  walk the list to find the proper ordered insertion point
                          --  following local variables track position of the search
                          INT           cursor,
                                        prior :
                          }}}
                          SEQ
                            {{{  initialize
                            prior  := head
                            cursor := node.awaits[head][3]
                            }}}
                  {{{  scan list
                  WHILE  cursor <> NIL
                    IF
                      count.value <= node.awaits[cursor][0]
                        cursor := NIL
                      ELSE
                        SEQ
                          prior  := cursor
                          cursor := node.awaits[cursor][3]
                  }}}
                  {{{  insert into list
                  wait.next := node.awaits[prior][3]
                  node.awaits[prior][3] := free.list[next.free]
                  }}}
                }}}
            {{{  build table entry - remove position from free list
            wait.count  := count.value
            wait.node   := from.node
            wait.proc   := from.proc
            next.free   := next.free + 1
            }}}
        }}}
    }}}
}}}
{{{  ticket command
action.code = TICKET.CMD
  -- transform ticket value to byte array for transmission as data
  []BYTE out.count RETYPES current.ticket :
  -- send packet and increment the ticket value
  SEQ
    send.packet([TICKET.REP, count.id, from.node, from.proc, node.id, 0],
                 SIZE out.count, out.count)
    current.ticket := current.ticket + 1
}}}
{{{  put command
action.code = PUT.CMD
  {{{  local definitions
  -- define the location in the node.data array of shared memory
  -- for where to write the transferred data
  VAL INT     put.base     IS       base + index :
  VAL []BYTE  index.array  RETYPES  index :
  VAL INT     index.size   IS       SIZE index.array :
  VAL INT     put.size     IS       data.size - index.size :
  }}}
  -- store the requested data array
  [node.data FROM put.base FOR put.size] :=
      [data.array FROM index.size FOR put.size]
}}}
{{{  get command
action.code = GET.CMD
  -- determine the starting location for the read operation
  -- in the shared memory segment
  VAL INT  get.base   IS  base + index :
  -- send the requested data
  send.packet([GET.REP, count.id, from.node, from.proc, node.id, 0],
                        seg.size, [node.data FROM get.base FOR seg.size])
        }}}
      }}}
      {{{  else  -- just pass the message on to its destination node
      ELSE
        send.packet(header, data.size, data.array)
      }}}

:
-----------------------------------------------------------------------
}}}
{{{  PROC build.packet(link.no)
-----------------------------------------------------------------------
PROC build.packet(VAL INT link.no)
-----------------------------------------------------------------------
{{{  description
-- To reduce the internal node communication load, message headers from local
-- processes include only a subset of the elements used for communicating
-- between nodes.  This procedure 'expands' a local communications packet
-- header into an external packet header.
}}}
-----------------------------------------------------------------------

  SEQ
    from.node := node.id
    from.proc := link.no
    to.node   := count.node[count.id]
    to.proc   := 0

:
-----------------------------------------------------------------------
}}}
}}}

SEQ
  {{{  initialization
  {{{  initialize the count.array and count.array.indices
  SEQ i = 0 FOR MAX.COUNTS
    count.array.indices[i] := NIL

  node.counts := 0
  base.count  := 0
  SEQ i = 0 FOR num.counts
    IF
      count.node[i] = node.id
        SEQ
          -- enter quick look-up index
          count.array.indices[i] := node.counts
          -- build initial count data
          count.array[node.counts][0] := 0
          count.array[node.counts][1] := 0
          count.array[node.counts][2] := base.count
          count.array[node.counts][3] := NIL
          -- allocate shared memory segment
          node.counts := node.counts + 1
          base.count := base.count + count.size[i]
        ELSE
          SKIP
    }}}
    {{{  initialize the free list
    SEQ i = 0 FOR MAX.AWAIT.QUE.ENTRIES
      free.list[i] := i
    next.free := 0
    }}}
    }}}
    {{{  run system
    PAR
      {{{  buffer hardware links
      PAR link.no = 0 FOR no.h.links
        PAR
          buffer.in.link(hard.links[link.no + no.h.links], links[link.no][INPUT])
          buffer.out.link(links[link.no][OUTPUT], hard.links[link.no])
      }}}
      {{{  monitor communications
      {{{  local declarations
      -- The communications monitoring procedure uses a 'fair' implementation
      -- of the ALT structure.  In general, it provides that if a communication
      -- was just received from one of several channels, that channel will have
      -- the lowest priority for the next execution of the ALT.  In this way,
      -- no single communications channel cam 'starve' access to the kernel
      -- from the other communications channels.  The variables defined
      -- below are used to track the last communicating channel for this
      -- 'fair' ALT.  Note that hard and soft links are treated separately.

      INT last.h, last.s:
      }}}
      SEQ
        {{{  initialization
        last.s := 0
        last.h := 0
        }}}
        WHILE TRUE
          PRI ALT
            {{{  handle external hardware links
            ALT link.no = 0 FOR no.h.links
              links[(link.no + last.h)\no.h.links][INPUT] ? header;
data.size::data.array
                SEQ
                  process.packet((link.no + last.h)\no.h.links)
                  last.h := (no.h.links - 1) + link.no
            }}}
            {{{  handle local soft links
            ALT i = 0 FOR no.s.links
              links[((i + last.s)\no.s.links) + no.h.links][INPUT] ? short.header;
                                                        data.size::data.array
                VAL INT  link.no IS ((i + last.s)\no.s.links) + no.h.links :
                SEQ
                  build.packet(link.no)
                  process.packet(link.no)
                  last.s := i + (no.s.links - 1)
            }}}
        }}}
    }}}

:
--------------------------------------------------------------------------------
```

C. READ PROCEDURE

```
--------------------------------------------------------------------------------
PROC read([2]CHAN OF ANY link,
          VAL INT        count.id,
          INT            count.value)
--------------------------------------------------------------------------------
{{{  description
--  This procedure reads and returns the value of the argument specified
--  event count.
}}}
--------------------------------------------------------------------------------

  {{{  libraries
  #USE "\ecslib\ecssymb.tsr"
  }}}

  {{{  declarations
  -- alternalte channel names
  CHAN OF ANY  linkin      IS  link[0] :
  CHAN OF ANY  linkout     IS  link[1] :

  -- communications data structures
  [2]INT       short.header :
  INT          data.size :
  []BYTE       data.array   RETYPES  count.value:
  }}}

  SEQ
    -- request count value from kernel
    linkout  !  [READ.CMD, count.id]; 0
    -- receive count value from kernel
    linkin   ?  short.header; data.size::data.array

:
--------------------------------------------------------------------------------
```

D. ADVANCE PROCEDURE

```
--------------------------------------------------------------------------------
PROC advance([2]CHAN OF ANY link,
             VAL INT        count.id)
--------------------------------------------------------------------------------
{{{  description
--  This procedure requests that the distributed kernel
--  increment the specified event counter.
}}}
--------------------------------------------------------------------------------

  {{{  libraries
  #USE "\ecslib\ecssymb.tsr"
  }}}

  {{{  declarations
  -- alternate name for channel
  CHAN OF ANY  linkout      IS  link[1] :
  }}}

  -- request the advance action
  linkout  !  [ADVANCE.CMD, count.id]; 0

:
--------------------------------------------------------------------------------
```

E. AWAIT PROCEDURE

```
--------------------------------------------------------------------------------
PROC await([2]CHAN OF ANY link,
           VAL INT        count.id,
           VAL INT        count.value)
--------------------------------------------------------------------------------
{{{  description
--  Perform the await function on the specified event count and
--  await the argument count value.

--  Note that the waiting process table (node.awaits) in the
--  kernel that maintains the event count may be full.
--  In this case, the kernel returns a message to this process
--  identifying that this is the case.  This procedure then
--  waits a period of time and retransmits the await request to
--  the kernel.  If the await request is again rejected due to
--  a full waiting process table, the wait time is doubled before
--  retrying the await request.  This process will continue until
--  either the requested wait-for count is reached or the await
--  request is accepted and placed in the kernel's waiting process
--  table.
}}}
--------------------------------------------------------------------------------

  {{{  libraries
  #USE "\ecslib\ecssymb.tsr"
  }}}

  {{{  declarations
  -- alternate channel names
  CHAN OF ANY  linkin       IS  link[0] :
  CHAN OF ANY  linkout      IS  link[1] :

  -- communications data structures
  [2]INT        short.header :
  INT           return.tag   IS       short.header[0] :
  VAL []BYTE    data.array   RETYPES  count.value :
  INT           dummy.size :
  [4]BYTE       dummy.array :

  -- structures for controlling retransmit of rejected awaits
  TIMER         clock :
  INT           current.time, wait.time :
  }}}

  SEQ
    -- request await from kernel
    linkout  !  [AWAIT.CMD, count.id]; (SIZE data.array)::data.array
    -- receive reply from the kernel
    linkin   ?  short.header; dummy.size::dummy.array
    {{{  check to see if the await was accepted
    IF
      return.tag = AWAIT.QUE.FULL
        {{{  wait by binary back-off and retry the await request
        SEQ
          -- initial wait time (short)
          wait.time  := 1
          WHILE return.tag = AWAIT.QUE.FULL
            SEQ
              clock    ?  current.time
              clock    ?  AFTER (current.time PLUS wait.time)
              -- set-up the doubled delay time
              wait.time := wait.time * 2
              -- request await from kernel
              linkout  !  [AWAIT.CMD, count.id]; (SIZE data.array)::data.array
              -- receive response from kernel
              linkin   ?  short.header; dummy.size::dummy.array
        }}}
      ELSE
        {{{  do nothing -- await was accepted
        SKIP
        }}}
    }}}

:
-----------------------------------------------------------------------------
```

F. TICKET PROCEDURE

```
-----------------------------------------------------------------------------
PROC ticket([2]CHAN OF ANY link,
            VAL INT       count.id,
            INT           count.value)
-----------------------------------------------------------------------------
{{{  description
--  This procedure request a reservation or 'ticket' from the
--  distributed kernel for the specified event count/sequencer.
--  The value of the ticket is returned in the count.value
-- parameter.
}}}
--------------------------------------------------------------------------------

  {{{ libraries
  #USE "\ecslib\ecssymb.tsr"
  }}}

  {{{ declarations
  -- alternate channel names
  CHAN OF ANY  linkin       IS  link[0] :
  CHAN OF ANY  linkout      IS  link[1] :

  -- communication data structures
  [2]INT       short.header :
  INT          data.size :
  []BYTE       data.array   RETYPES  count.value :
  }}}

  SEQ
    -- request ticket from kernel
    linkout  !  [TICKET.CMD, count.id]; 0
    -- receive ticket from kernel
    linkin   ?  short.header; data.size::data.array

:
--------------------------------------------------------------------------------
```

G. PUT PROCEDURE

```
--------------------------------------------------------------------------------
PROC put([2]CHAN OF ANY link,
         VAL INT        count.id,
         VAL INT        index,
         VAL []BYTE     data.array)
--------------------------------------------------------------------------------
{{{ description
--  This procedure writes data to the shared memory segment
--  associated with the argument event count.  The data array
--  is written offset from the start of the shared memory
--  segment by the number bytes specified by the index
--  parameter.
}}}
--------------------------------------------------------------------------------

  {{{ libraries
  #USE "\ecslib\ecssymb.tsr"
  }}}

  {{{ declarations
  -- alternate channel name
  CHAN OF ANY  linkout      IS      link[1] :

  -- convert the index value to an array of bytes for
  -- transmission as data
  VAL []BYTE    index.array  RETYPES  index :

  -- identify the size of the data arrays
  VAL INT       index.size   IS       SIZE index.array :
  VAL INT       data.size    IS       SIZE data.array :
  VAL INT       local.size   IS       index.size + data.size :

  -- define a local array for the data and index value
  [MAX.MESSAGE.SIZE + index.size]BYTE     local.array :
  }}}

  SEQ
    -- combine the data array and the array representation of
    -- the index value into a single array
    [local.array FROM 0 FOR index.size] := index.array
    [local.array FROM index.size FOR data.size] := data.array
    -- request kernel to store the data array
    linkout  !  [PUT.CMD, count.id]; local.size::local.array

:
--------------------------------------------------------------------------------
```

H. GET PROCEDURE

```
--------------------------------------------------------------------------------
PROC get([2]CHAN OF ANY link,
         VAL INT        count.id,
         VAL INT        index,
         []BYTE         data.array)
--------------------------------------------------------------------------------
{{{  description
--  This procedure performs a read of the shared memory
--  segment associated with the argument event count.
--  The number of bytes read is determined by the size
--  of the argument data array.  The bytes read from
--  the shared memory are offset from the start of the
--  shared memory segment by the number of bytes specified
--  in the index parameter.
}}}
--------------------------------------------------------------------------------

  {{{  libraries
  #USE "\ecslib\ecssymb.tsr"
  }}}

  {{{  declarations
  -- alternate name for the channels to kernel
  CHAN OF ANY  linkin       IS       link[0] :
  CHAN OF ANY  linkout      IS       link[1] :

  -- communications data structures
  [2]INT       short.header :
  INT          data.size :
```

```
  -- define the size and index retrieval parameters as
  -- an array of bytes for sending as data
  VAL []BYTE   get.specs    RETYPES  [index, SIZE data.array] :
  VAL INT      spec.size    IS       SIZE get.specs :
  }}}

  SEQ
    -- request the shared memory read operation
    linkout  !  [GET.CMD, count.id]; spec.size::get.specs
    -- receive the results of the read operation
    linkin   ?  short.header; data.size::data.array

:
--------------------------------------------------------------------------------
```

APPENDIX D

SAMPLE PROGRAM USING THE SHARED-MEMORY INTERFACE

A. DESCRIPTION

This appendix provides an example of the methodology employed to write a program using the shared-memory interface developed for this thesis. The programming example selected to demonstrate the methodology is the bounded buffer problem. In this problem, a producer passes data to a consumer via a bounded buffer or queue. The buffer serves to "smooth out" variations in the data production and consumption rates. To add slightly to the single producer bounded buffer problem, this implementation of the problem will provide for two producers of data.

B. MODULARIZATION

This particular problem can be naturally subdivided into three basic modules. Each of these are listed and described below.

1. Producers

Each instantiation of this module generates a continuous stream of data elements at a specified average rate with some characteristic random variation in the rate. The data elements produced are placed in a segment of memory shared with the consumer of the data. Access to the shared-memory segment is controlled using an event count and a sequencer. The sequencer controls the access of the two producers to the buffer. The associated event count is advanced when either producer uses its ticket to add data to the buffer.

2. Consumer

This module continuously removes data elements from the shared memory buffer at a specified average rate with some characteristic random variation in the rate. The average rate for the single consumer should be at least equal to the total of the producer's rates. If not and the consumer can not keep up with the producers, the buffer will eventually fill and the producers will be forced to remain idle while waiting for the consumer.

An event count associated with the consumer is advanced when a data element is removed from the buffer. Note that if the producer and consumer event counts are initially equal, the number of data elements in the buffer will be equal to the difference between the event counts.

C. MODULE CODING

The code for each of the program modules should then be developed. The following sections provide an abbreviated listing of the code for the modules.

1. Producers

```
------------------------------------------------------------
PROC producer([2]CHAN OF ANY link,
              VAL INT        in, out)
------------------------------------------------------------

  #USE "\ecslib\ecsproc.tsr"   --  the interface library
  #USE "globals.tsr"           --  global constants

  INT                          my.ticket :
```

```
  [data.element.size]BYTE      data.element :

  WHILE TRUE
    SEQ

      --  insert code to produce a data element

      ticket(link, in, my.ticket)
      await(link, in, my.ticket)
      await(link, out,
        (my.ticket - data.buffer.size) + 1)
      put(link, in,
        (my.ticket + 1)\data.buffer.size, data.element)
      advance(link, out)

:
-----------------------------------------------------------
```

2. Consumer

```
-----------------------------------------------------------
PROC consumer([2]CHAN OF ANY  link,
              VAL INT         in, out)
-----------------------------------------------------------

  #USE "\ecslib\ecsproc.tsr"   --  the interface library
  #USE "globals.tsr"           --  global constants

  INT                          my.count :
  [data.element.size]BYTE      data.element :

  SEQ
    my.count := 0
    WHILE TRUE
      SEQ
        my.count := my.count + 1
        await(link, in, my.count)
        get(link, out,
          my.count\data.buffer.size, data.element)
        advance(link, out)

        --  insert code for consuming the data element

:
-----------------------------------------------------------
```

D. APPORTIONMENT OF MODULES

To best demonstrate the nature of the abstracted programming interface, the apportionment of modules will be done in two different ways. The first way will assign all modules, shared memory, and event counts to a single Transputer. The second way will distribute the modules, shared memory, and event counts on three different Transputers.

1. Single Transputer

```
------------------------------------------------------------
Library aport.tsr
------------------------------------------------------------
#USE "globals.tsr"          -- global constants

-- symbolic constants for counts
VAL  in             IS   0 :
VAL  out            IS   1 :

-- count node assignments (which node maintains the count)
VAL  count.node     IS   [0, 0] :

-- shared memory segment sizes
VAL  count.size     IS   [0, data.buffer.size] :

-- network adjacency matrix to match particular
-- physical configuration.  This matrix matches
-- a clockwise ring on a B003 board.
VAL  node.link      IS   [[0,2,2,2],
                          [2,0,2,2],
                          [2,2,0,2],
                          [2,2,2,0]] :
------------------------------------------------------------
------------------------------------------------------------
PROC node0() -- procedure to run on Transputer 0
------------------------------------------------------------

  #USE "\ecslib\ecsproc.tsr"   -- the interface library
  #USE "procs.tsr"             -- library of procedures
  #USE "aport.tsr"             -- apportionment structures

  VAL INT           node.id      IS  0 :
  [8][2]CHAN OF ANY links :
  CHAN OF ANY       pro.one.link IS  links[4] :
  CHAN OF ANY       pro.two.link IS  links[5] :
  CHAN OF ANY       consume.link IS  links[6] :

  PRI PAR
    ecs.kernel(links, node.id,
               count.node, count.size,
               node.link[node.id])
    PAR
      producer(pro.one.link, in, out)
      producer(pro.two.link, in, out)
      consumer(consume.link, in, out)
```

```
:
------------------------------------------------------------
```

2. Three Transputers

```
------------------------------------------------------------
Library aport.tsr
------------------------------------------------------------
#USE "globals.tsr"          -- global constants

-- symbolic constants for counts
VAL  in              IS   0 :
VAL  out             IS   1 :

-- count node assignments (which node maintains the count)
VAL  count.node      IS   [0, 2] :

-- shared memory segment sizes
VAL  count.size      IS   [0, data.buffer.size] :

-- network adjacency matrix to match particular
-- physical configuration.  This matrix matches
-- a clockwise ring on a B003 board.
VAL  node.link       IS   [[0,2,2,2],
                           [2,0,2,2],
                           [2,2,0,2],
                           [2,2,2,0]] :
------------------------------------------------------------
------------------------------------------------------------
PROC node0() --  procedure to run on Transputer 0
------------------------------------------------------------

  #USE "\ecslib\ecsproc.tsr"   --  the interface library
  #USE "procs.tsr"             --  library of procedures
  #USE "aport.tsr"             --  apportionment structures

  VAL INT             node.id        IS  0 :
  [5][2]CHAN OF ANY   links :
  CHAN OF ANY         pro.one.link   IS  links[4] :

  PRI PAR
    ecs.kernel(links, node.id,
               count.node, count.size,
               node.link[node.id])
    producer(pro.one.link, in, out)

:
------------------------------------------------------------
------------------------------------------------------------
PROC node1() --  procedure to run on Transputer 1
------------------------------------------------------------

  #USE "\ecslib\ecsproc.tsr"   --  the interface library
  #USE "procs.tsr"             --  library of procedures
  #USE "aport.tsr"            --  apportionment structures

  VAL INT            node.id       IS  1 :
  [5][2]CHAN OF ANY  links :
  CHAN OF ANY        pro.two.link  IS  links[4] :

  PRI PAR
    ecs.kernel(links, node.id,
               count.node, count.size,
               node.link[node.id])
    producer(pro.two.link, in, out)

:
------------------------------------------------------------
------------------------------------------------------------
PROC node2() --  procedure to run on Transputer 2
------------------------------------------------------------

  #USE "\ecslib\ecsproc.tsr"  --  the interface library
  #USE "procs.tsr"            --  library of procedures
  #USE "aport.tsr"            --  apportionment structures

  VAL INT            node.id       IS  2 :
  [5][2]CHAN OF ANY  links :
  CHAN OF ANY        consume.link  IS  links[4] :

  PRI PAR
    ecs.kernel(links, node.id,
               count.node, count.size,
               node.link[node.id])
    consumer(consume.link, in, out)

:
------------------------------------------------------------
```

APPENDIX E

DETAILED SOURCE CODE FOR THE MESSAGE-PASSING ABSTRACT INTERFACE LIBRARY

A. SYMBOLIC CONSTANTS

```
{{{  LIB
...  Library ID
{{{  VAL declarations
{{{  useful symbolic constants
VAL    ELSE                IS  TRUE :
VAL    NIL                 IS  -1 :
VAL    SENDER              IS  0 :
VAL    OUTPUT              IS  0 :
VAL    RECEIVER            IS  1 :
VAL    INPUT               IS  1 :
}}}
{{{  system limits
--  define the maximum communications data array size
VAL    MAX.MESSAGE.SIZE    IS  1024 :

--  define the maximum number of system nodes
VAL    MAX.NODES           IS  16 :

--  define the maximum number of global channels
VAL    MAX.CHANS           IS  256 :

--  define the mazimum number of channels per node
VAL    MAX.PROC.PER.NODE   IS  40 :

--  define the mazimum number of processes per node
VAL    MAX.CHAN.PER.NODE   IS  40 :
}}}
{{{  packet tags used during initialization
VAL    INIT.START          IS  1 :
VAL    START.ACK           IS  2 :

VAL    INIT.DATA           IS  3 :
VAL    DATA.ACK            IS  4 :

VAL    INIT.STOP           IS  5 :
VAL    STOP.ACK            IS  6 :

VAL    INIT.QUIT           IS  9 :
}}}
{{{  packet tags used during communication
VAL    RECEIVER.READY      IS  7 :

VAL    SEND                IS  8 :
}}}
}}}
}}}
```

B. KERNEL PROCEDURE

```
--------------------------------------------------------------------------------
PROC csp.kernel(VAL INT          node.id,
                VAL []INT        node.link,
                VAL [][2]INT     chan.map,
                []CHAN OF ANY    loc.chan)
-------------------------------------------------------------------------------
{{{  description
--  This procedure is the distributed kernel for a message passing
--  based model programming interface.  This procedure should be
--  executed in parallel with application program modules using the
--  interface.  This procedure should be run at high priority and
--  the application modules run at low priority.
}}}
-------------------------------------------------------------------------------

  {{{  libraries
  #USE "cspsymb.tsr"
  }}}

  {{{  declarations
  {{{  local abbreviations
  VAL INT   no.h.links    IS     4 :
  VAL INT   no.s.links    IS     SIZE chan.map :
  VAL INT   no.l.chan     IS     SIZE chan.map :
  VAL INT   no.links      IS     no.h.links + no.s.links :
  VAL INT   num.nodes     IS     SIZE node.link :
  }}}
  {{{  channel declaration and placement
  [no.h.links*2]CHAN OF ANY  hard.links:
  PLACE hard.links AT 0 :

  [MAX.CHAN.PER.NODE][2]CHAN OF ANY  links:
  }}}
  {{{  communications data structures
  -- definition of the communications header parts
  [3]INT     header :
  INT        header.action.code    IS       header[0] :
  INT        header.to.node        IS       header[1] :
  INT        header.gchan          IS       header[2] :

  -- definition of the communications packet data segment
  INT                          data.size :
  [MAX.MESSAGE.SIZE]BYTE       data.array :
  }}}
  {{{  channel mapping data structures
  -- kernel data structures for communications routing and
  -- management
  [MAX.CHANS][2]INT  gchan.node,
                   gchan.lchan :
[MAX.CHANS]INT     lchan.gchan :
}}}
}}}

{{{  procedures
{{{  PROC get.local(local, return)
--------------------------------------------------------------------------------
PROC get.local(CHAN OF ANY  local, return)
--------------------------------------------------------------------------------
{{{  description
--  This procedure broadcasts local channel map data
--  to all nodes for network global structure
--  initialization
}}}
--------------------------------------------------------------------------------

  {{{  declarations
  -- for receiving acknowledge of remote broadcast
  INT     acknowledge :
  }}}

  SEQ
    {{{  broadcast start signals
    -- Open a path to all nodes in the system.
    -- Intermediate nodes receiving these start
    -- tokens and relaying them will not shut-down
    -- until a corresponding stop token is received.
    SEQ to.node = 0 FOR num.nodes
      IF
        to.node <> node.id
          SEQ
            local   !  [INIT.START, to.node, node.id]; 0
            return  ?  acknowledge
        ELSE
          SKIP
    }}}
    {{{  broadcast all local link data
    --  scan the channel map and build the local structures
    SEQ chan.no = 0 FOR no.l.chan
      {{{  local abbreviations
      VAL []INT   global.ident     RETYPES  chan.map[chan.no] :
      VAL []BYTE  ident.array      RETYPES  chan.map[chan.no] :
      VAL INT     global.chan.id   IS       global.ident[0] :
      VAL INT     chan.mode        IS       global.ident[1] :
      VAL INT     link.no          IS       chan.no + no.h.links :
      }}}
      SEQ
        -- build local data structures
        gchan.node[global.chan.id][chan.mode] := node.id
        gchan.lchan[global.chan.id][chan.mode] := link.no
        lchan.gchan[link.no] := global.chan.id
        -- broadcast to other nodes
        SEQ to.node = 0 FOR num.nodes
          IF
          to.node <> node.id
            SEQ
              local   !  [INIT.DATA, to.node, node.id]; 8::ident.array
              return  ?  acknowledge
          ELSE
            SKIP
    }}}
    {{{  broadcast done signals
    -- notify other nodes that done with init transmission
    SEQ to.node = 0 FOR num.nodes
      IF
        to.node <> node.id
          SEQ
            local   !  [INIT.STOP, to.node, node.id]; 0
            return  ?  acknowledge
        ELSE
          SKIP
    }}}
    {{{  send local quit signal
    -- notify that local transmissions over
    local ! [INIT.QUIT, node.id,node.id]; 0
    }}}

:
--------------------------------------------------------------------------------
}}}
{{{  PROC get.remote(remote, return)
--------------------------------------------------------------------------------
PROC get.remote(CHAN OF ANY  remote, return)
--------------------------------------------------------------------------------
{{{  description
--  This procedure inputs remote nodes' channel map data
--  to all nodes for network global structure
--  initialization
}}}
--------------------------------------------------------------------------------

  {{{  local declarations
  [3]INT     header :
  INT        init.code          IS      header[0]  :
  INT        to.node.id         IS      header[1]  :
  INT        from.node.id       IS      header[2]  :

  INT        data.size :
  [2]INT     global.ident :
  []BYTE     ident.array      RETYPES  global.ident :
  INT        global.chan.id   IS       global.ident[0]  :
  INT        chan.mode        IS       global.ident[1]  :

  INT        quit.count :
  }}}

  SEQ
    {{{  initialize
    -- will receive at least 2 messages from every other node
```

```
quit.count := 2*(num.nodes - 1)
}}}
WHILE quit.count > 0
  ALT link.no = 0 FOR no.h.links
    -- get a message from somewhere
    links[link.no][INPUT] ? header; data.size::ident.array
      IF
        {{{  for my node, process it
        to.node.id = node.id
          SEQ
            to.node.id := from.node.id
            -- categorize and act on message
            IF
              {{{  start message
              init.code = INIT.START
                SEQ
                  -- acknowledge message to remote node
                  init.code := START.ACK
                  remote ! header; data.size::ident.array
              }}}
              {{{  start ack
              init.code = START.ACK
                -- let local process know the ack rec'd
                return ! NIL
              }}}
              {{{  data message
              init.code = INIT.DATA
                SEQ
                  gchan.node[global.chan.id][chan.mode]  := from.node.id
                  -- acknowledge message to remote node
                  init.code := DATA.ACK
                  remote ! header; data.size::ident.array
              }}}
              {{{  data ack
              init.code = DATA.ACK
                -- let local process know the ack rec'd
                return ! NIL
              }}}
              {{{  stop message
              init.code = INIT.STOP
                SEQ
                  -- one less message to get from a remote node
                  quit.count := quit.count - 1
                  -- acknowledge message to remote node
                  init.code := STOP.ACK
                  remote ! header; data.size::ident.array
              }}}
              {{{  stop ack
              init.code = STOP.ACK
                SEQ
                  -- one less message to get from a remote node
                  quit.count := quit.count - 1
                  -- let local process know the ack rec'd
                  return ! NIL
              }}}
          }}}
          {{{  for a different node, forward it
          ELSE
            SEQ
              remote ! header; data.size::ident.array
              IF
                init.code = INIT.START
                  -- reserve a path for the sender
                  quit.count := quit.count + 1
                init.code = START.ACK
                  SKIP
                init.code = INIT.DATA
                  SKIP
                init.code = DATA.ACK
                  SKIP
                init.code = INIT.STOP
                  SKIP
                init.code = STOP.ACK
                  -- done with pass-thru reservation
                  quit.count := quit.count - 1
                ELSE
                  SKIP
          }}}
    {{{  send local quit signal
    -- done with remote receives
    remote ! [INIT.QUIT, node.id, node.id]; 0
    }}}

:
------------------------------------------------------------------------------
}}}
{{{  PROC multiplex(local, remote)
------------------------------------------------------------------------------
PROC  multiplex(CHAN OF ANY local, remote)
------------------------------------------------------------------------------
{{{  description
--  This procedure multiplexes locally generated initialization
--  messages and 'pass-through' initilization messages onto
--  the hardware communication links
}}}
------------------------------------------------------------------------------

  {{{  local declarations
  -- packet header definition
  [3]INT     header :
  INT        init.code           IS      header[0] :
  INT        to.node.id          IS      header[1] :

  -- init packet data
  INT        data.size :
  [2]INT     global.ident :
  []BYTE     ident.array      RETYPES  global.ident :

  -- local flags
  BOOL    local.done,
                remote.done :
        }}}

        SEQ
          {{{  initialize
          local.done := FALSE
          remote.done := FALSE
          }}}
          WHILE (NOT local.done) OR (NOT remote.done)
            PRI ALT
              {{{  remotely generated messages
              (NOT remote.done) & remote ? header; data.size::ident.array
                IF
                  init.code = INIT.QUIT
                    remote.done := TRUE
                  ELSE
                    links[node.link[to.node.id]][OUTPUT] ! header;
data.size::ident.array
              }}}
              {{{  locally generated messages
              (NOT local.done) & local ? header; data.size::ident.array
                IF
                  init.code = INIT.QUIT
                    local.done := TRUE
                  ELSE
                    links[node.link[to.node.id]][OUTPUT] ! header;
data.size::ident.array
              }}}

      :
      --------------------------------------------------------------------------
      }}}
      {{{  PROC map.out(local.link,local.chan)
      --------------------------------------------------------------------------
      PROC map.out([2]CHAN OF ANY  local.link,
                   CHAN OF ANY     local.chan)
      --------------------------------------------------------------------------
      {{{  description
      --  Perform the data transfers needed to connect the output
      --  end of a global channel to the receiving local channel
      }}}
      --------------------------------------------------------------------------

        {{{  declarations
        -- alternate name for the local bidirectional channel
        CHAN OF ANY    link.in     IS  local.link[0] :
        CHAN OF ANY    link.out    IS  local.link[1] :

        -- for receiving signal to start transmission
        INT                            any :

        -- data structure for transmitted message
        INT                            data.size :
        [MAX.MESSAGE.SIZE]BYTE         data.array :
        }}}
```

```
  WHILE TRUE
    SEQ
      PAR
        -- know that receiver is ready to receive
        link.in  ?  any
        -- get the transmitted data
        local.chan  ?  data.size::data.array
      -- send the data via the kernel
      link.out  !  SEND; data.size::data.array

:
--------------------------------------------------------------------------------
}}}
{{{  PROC map.in (local.link,local.chan)
--------------------------------------------------------------------------------
PROC map.in([2]CHAN OF ANY  local.link,
             CHAN OF ANY    local.chan)
--------------------------------------------------------------------------------
{{{  description
--  Perform the data transfers needed to connect the input
--  end of a global channel to the transmitting local channel
}}}
--------------------------------------------------------------------------------

  {{{  declarations
  -- alternate names for the local channels
  CHAN OF ANY    link.in    IS  local.link[0] :
  CHAN OF ANY    link.out   IS  local.link[1] :

  -- data structure for received data
  INT                          data.size :
  [MAX.MESSAGE.SIZE]BYTE       data.array :
  }}}

  WHILE TRUE
    SEQ
      -- identify to the kernel that the process is ready to receive
      link.out  !  RECEIVER.READY; 0
      -- receive the transmitted data from the kernel
      link.in   ?  data.size::data.array
      -- send data to the application
      local.chan  !  data.size::data.array

:
--------------------------------------------------------------------------------
}}}
{{{  PROC buffer.link(chan.in,chan.out)
--------------------------------------------------------------------------------
PROC buffer.link(CHAN OF ANY in.link, out.link)
--------------------------------------------------------------------------------
{{{  description
-- This procedure provides a buffer for hard link input.
-- This buffer increases the overall throughput of the node.
}}}
```

```
--------------------------------------------------------------------------------

  {{{  declarations
  [MAX.MESSAGE.SIZE]BYTE          data.array.0,
                                  data.array.1 :
  [3]INT                          header.0,
                                  header.1 :
  INT                             data.size.0,
                                  data.size.1 :
  }}}

  SEQ
    in.link   ?  header.0; data.size.0::data.array.0
    WHILE TRUE
      SEQ
        PAR
          out.link  !  header.0; data.size.0::data.array.0
          in.link   ?  header.1; data.size.1::data.array.1
        PAR
          out.link  !  header.1; data.size.1::data.array.1
          in.link   ?  header.0; data.size.0::data.array.0

:
--------------------------------------------------------------------------------
}}}
{{{  PROC build packet(link.no)
--------------------------------------------------------------------------------
PROC build.packet(VAL INT link.no)
--------------------------------------------------------------------------------
{{{  description
--  Based on the local channel sending the message to the kernel,
--  construct an internode communications packet.
}}}
--------------------------------------------------------------------------------

  SEQ
    -- identify the global channel being used
    header.gchan      := lchan.gchan[link.no]
    -- find the node to which the packet is to be routed
    IF
      header.action.code = RECEIVER.READY
        header.to.node   := gchan.node[header.gchan][OUTPUT]
      header.action.code = SEND
        header.to.node   := gchan.node[header.gchan][INPUT]
      ELSE
        header.to.node   := node.id

:
--------------------------------------------------------------------------------
}}}
{{{  PROC process.packet(link.no)
--------------------------------------------------------------------------------
PROC process.packet(VAL INT link.no)
--------------------------------------------------------------------------------
{{{  description
```

```
        --  This procedure performs communications routing and maintains
        --  synchronization between the sending and receiving processes
        --  by sequencing communications between the local and remote
        --  processes.
        }}}
        --------------------------------------------------------------------------------

          IF
            {{{  packet for this node
            header.to.node = node.id
              IF
                header.action.code = RECEIVER.READY
                  -- let the sender know that receiver is ready
                  links[gchan.lchan[header.gchan][OUTPUT]][OUTPUT] ! SEND
                header.action.code = SEND
                  -- pass on the data packet to its destination
                  links[gchan.lchan[header.gchan][INPUT]][OUTPUT] !
data.size::data.array
            }}}
            {{{  packet for other node - forward it
            ELSE
              links[node.link[header.to.node]][OUTPUT] ! header; data.size::data.array
            }}}

        :
        --------------------------------------------------------------------------------
        }}}
        }}}

        PAR
          {{{  buffer hardware links
          PAR link.no = 0 FOR no.h.links
            PAR
              buffer.link(hard.links[link.no + no.h.links], links[link.no][INPUT])
              buffer.link(links[link.no][OUTPUT], hard.links[link.no])
          }}}
          SEQ
            {{{  initialization
            {{{  initialize kernel data structures
            SEQ i = 0 FOR MAX.CHANS
              SEQ
                lchan.gchan[i]  := NIL
                gchan.node[i]   := [NIL,NIL]
                gchan.lchan[i]  := [NIL,NIL]
            }}}
            {{{  query local channels to build data structures
            {{{  declare local channels for initialization
            CHAN OF ANY    local,
                           remote,
                           return :
            }}}
            PAR
              get.local(local,return)
              get.remote(remote, return)
              multiplex(local, remote)
```

```
}}}
}}}
PAR
  {{{  build mapping processes
  --  Create the channel controller processes to connect the
  --  local channels to global channel ends
  {{{  local declarations
  TIMER                clock:
  INT                  time:
  }}}
  SEQ
    {{{  wait for init messages clear the network
    --  This is a cobble to correct a problem with one
    --  node completinig initialization and sending a 'real'
    --  message that is not properly handled by a node
    --  that has not yet been initialized.  Just wait for
    --  a long time for the initilization messages to
    --  clear the network
    clock ? time
    clock ? AFTER time + 5000000 -- wait 5 seconds
    }}}
    PAR chan.no = 0 FOR MAX.CHAN.PER.NODE
      IF -- this channel is among those declared
        chan.no < no.l.chan
          IF  -- input or output global channel end
            chan.map[chan.no][1] = INPUT
              map.in(links[chan.no+no.h.links],loc.chan[chan.no])
            ELSE
              map.out(links[chan.no+no.h.links],loc.chan[chan.no])
        ELSE
          SKIP
  }}}
  {{{  monitor communications
  --  accept input from channel controllers or from hard links
  {{{  local declarations
  -- The communications monitoring procedure uses a 'fair' implementation
  -- of the ALT structure.  In general, it provides that if a communication
  -- was just received from one of several channels, that channel will have
  -- the lowest priority for the next execution of the ALT.  In this way,
  -- no single communications channel cam 'starve' access to the kernel
  -- from the other communications channels.  The variables defined
  -- below are used to track the last communicating channel for this
  -- 'fair' ALT.  Note that hard and soft links are treated separately.

  INT last.h, last.s:
  }}}
  SEQ
    {{{  initialization
    last.s := 0
    last.h := 0
    }}}
    WHILE TRUE
      PRI ALT
        {{{  handle external hardware links
        ALT link.no = 0 FOR no.h.links
```

```
                links[(link.no + last.h)\no.h.links][INPUT] ? header;
                                                   data.size::data.array
                  SEQ
                    process.packet((link.no + last.h)\no.h.links)
                    last.h := (no.h.links - 1) + link.no
              }}}
              {{{  handle local soft links
              ALT i = 0 FOR no.s.links
                links[((i + last.s)\no.s.links) + no.h.links][INPUT] ?
header.action.code;
                                                   data.size::data.array
                  VAL INT  link.no IS ((i + last.s)\no.s.links) + no.h.links :
                  SEQ
                    build.packet(link.no)
                    process.packet(link.no)
                    last.s := i + (no.s.links - 1)
              }}}
          }}}

    :
    --------------------------------------------------------------------------
```

APPENDIX F

SAMPLE PROGRAM USING THE MESSAGE-PASSING INTERFACE

A. DESCRIPTION

This appendix provides an example of the methodology employed to write a program using the message-passing interface developed for this thesis. The programming example selected is the same bounded buffer problem used in Appendix D to demonstrate programming with the shared-memory interface. This problem consisted of two producers and one consumer with a bounded buffer or queue between them.

B. MODULARIZATION

The modularization of the bounded buffer problem for programming under the message-passing interface is similar to that specified for programming with the shared memory interface. As with the shared memory modularization, the producers and consumer are individual modules. However, with the shared memory interface, the buffer between the producers and consumer is a natural consequence of the memory shared between the modules. In the message-passing scheme, however, this buffer must be defined and coded as a separate module.

1. Producers

Each instantiation of this module generates a continuous stream of data elements at a specified average rate with some characteristic random variation in the rate. The data elements produced are output to a buffering process via a global communications channel.

2. Consumer

This module continuously attempts to input data elements from a global communications channel. Data elements input are consumed at a specified average rate with some characteristic random variation in the rate. The average rate for the single consumer should be at least equal to the total of the producer's rates. If not and the consumer can not keep up with the producers, the buffer will eventually fill and the producers will be forced to remain idle while waiting for the consumer.

3. Buffer

This module inputs communications from two global channels and queues the input data elements for output on a third global communications channel. Internally, this buffer should exhibit first-in, first-out queue characteristics.

C. MODULE CODING

The code for each of the program modules should then be developed. The following sections provide an abbreviated listing of the code for the modules.

1. Producers

```
-------------------------------------------------------------
PROC producer(CHAN OF ANY output)
-------------------------------------------------------------

  #USE "\csplib\cspproc.tsr"    -- the interface library
  #USE "globals.tsr"            -- global constants
  [data.element.size]BYTE      data.element :

  WHILE TRUE
    SEQ

      --  insert code for producing a data element

      output ! data.element

:
-----------------------------------------------------------
```

2. Consumer

```
-----------------------------------------------------------
PROC consumer(CHAN OF ANY  input)
-----------------------------------------------------------

  #USE "\csplib\cspproc.tsr"    --  the interface library
  #USE "globals.tsr"            --  global constants

  [data.element.size]BYTE      data.element :

  WHILE TRUE
    SEQ

      input  ?  data.element

      --  insert code to consume data element

:
-----------------------------------------------------------
```

3. Buffer

```
-----------------------------------------------------------
PROC buffer(CHAN OF ANY  buff.in.1, buff.in.2, buff.out)
-----------------------------------------------------------
--  Note, procedure written in a manner to illustrate some
--  features of the OCCAM programming language; not for
--  efficiency

  #USE "\csplib\cspproc.tsr"    --  the interface library
  #USE "globals.tsr"            --  global constants

  [buffer.size]CHAN OF ANY            buff.chan :

  PAR

    -- accept input from either producer
    [data.element.size]BYTE  data.element :
    WHILE TRUE
      ALT
        buff.in.1  ?  data.element
          buff.chan[0]  !  data.element
        buff.in.2  ?  data.element
          buff.chan[0]  !  data.element

    -- buffer the input
    PAR i = 0 FOR buffer.size - 1
      [data.element.size]BYTE  data.element :
      WHILE TRUE
        SEQ
          buff.chan[i]  ?  data.element
          buff.chan[i + 1]  !  data.element

    -- send from the buffer
    [data.element.size]BYTE  data.element :
    WHILE TRUE
      SEQ
        buff.chan[buffer.size - 1]  ?  data.element
        buff.out  !  data.element

:
------------------------------------------------------------
```

D. APPORTIONMENT OF MODULES

As is done for the shared-memory interface, the apportionment of modules for the message-passing interface is done in two different ways. The first way will assign all modules to a single Transputer. The second way will distribute the modules on four different Transputers.

1. Single Transputer

```
------------------------------------------------------------
Library aport.tsr
------------------------------------------------------------
#USE "globals.tsr"           -- global constants

--  symbolic constants for global channels
VAL  pro1.global     IS   0 :
VAL  pro2.global     IS   1 :
VAL  con.global      IS   2 :

--  network adjacency matrix to match particular
--  physical configuration.  This matrix matches
--  a clockwise ring on a B003 board.
VAL  node.link       IS   [[0,2,2,2],
                           [2,0,2,2],
                           [2,2,0,2],
                           [2,2,2,0]] :
------------------------------------------------------------
------------------------------------------------------------
PROC node0() --  procedure to run on Transputer 0
```

```
----------------------------------------------------------

  #USE "\csplib\cspproc.tsr"   --  the interface library
  #USE "procs.tsr"             --  library of procedures
  #USE "aport.tsr"             --  apportionment structures

  VAL INT           node.id       IS  0 :
  VAL [][2]INT      chan.map      IS  [[pro1.global,SENDER],
                                        [pro1.global,RECEIVER],
                                        [pro2.global,SENDER],
                                        [pro2.global,RECEIVER],
                                        [con.global,SENDER],
                                        [con.global,RECEIVER]] :

  [SIZE chan.map]CHAN OF ANY  loc.chan :
  --  abbreviations for convenience only
  CHAN OF ANY       pro.one.out   IS  loc.chan[0] :
  CHAN OF ANY       buff.in.one   IS  loc.chan[1] :
  CHAN OF ANY       pro.two.out   IS  loc.chan[2] :
  CHAN OF ANY       buff.in.two   IS  loc.chan[3] :
  CHAN OF ANY       buff.out      IS  loc.chan[4] :
  CHAN OF ANY       con.in        IS  loc.chan[5] :

  PRI PAR
    csp.kernel(node.id, node.link[node.id],
               chan.map, loc.chan)
    PAR
      producer(pro.one.out)
      producer(pro.two.out)
      buffer(buff.in.one, buff.in.two, buff.out)
      consumer(con.in)

:
----------------------------------------------------------
```

2. Four Transputers

```
----------------------------------------------------------
Library aport.tsr
----------------------------------------------------------
#USE "globals.tsr"            --  global constants

--  symbolic constants for global channels
VAL  pro1.global    IS   0 :
VAL  pro2.global    IS   1 :
VAL  con.global     IS   2 :

--  network adjacency matrix to match particular
--  physical configuration.  This matrix matches
--  a clockwise ring on a B003 board.
VAL  node.link      IS   [[0,2,2,2],
                           [2,0,2,2],
                           [2,2,0,2],
                           [2,2,2,0]] :
----------------------------------------------------------
```

```
-----------------------------------------------------------
PROC node0() --  procedure to run on Transputer 0
-----------------------------------------------------------

  #USE "\csplib\cspproc.tsr"   --  the interface library
  #USE "procs.tsr"             --  library of procedures
  #USE "aport.tsr"             --  apportionment structures

  VAL INT           node.id      IS  0 :
  VAL [][2]INT      chan.map     IS  [[pro1.global,SENDER]] :

  [SIZE chan.map]CHAN OF ANY  loc.chan :
  --  abbreviation for convenience only
  CHAN OF ANY       pro.one.out   IS  loc.chan[0] :

  PRI PAR
    csp.kernel(node.id, node.link[node.id],
               chan.map, loc.chan)
    producer(pro.one.out)

:
-----------------------------------------------------------
-----------------------------------------------------------
PROC node1() --  procedure to run on Transputer 1
-----------------------------------------------------------

  #USE "\csplib\cspproc.tsr"   --  the interface library
  #USE "procs.tsr"             --  library of procedures
  #USE "aport.tsr"             --  apportionment structures

  VAL INT           node.id      IS  1 :
  VAL [][2]INT      chan.map     IS  [[pro2.global,SENDER]]:

  [SIZE chan.map]CHAN OF ANY  loc.chan :
  --  abbreviations for convenience only
  CHAN OF ANY       pro.two.out   IS  loc.chan[0] :

  PRI PAR
    csp.kernel(node.id, node.link[node.id],
               chan.map, loc.chan)
    producer(pro.two.out)

:
-----------------------------------------------------------
-----------------------------------------------------------
PROC node2() --  procedure to run on Transputer 2
-----------------------------------------------------------

  #USE "\csplib\cspproc.tsr"   --  the interface library
  #USE "procs.tsr"             --  library of procedures
  #USE "aport.tsr"             --  apportionment structures

  VAL INT           node.id      IS  2 :
  VAL [][2]INT      chan.map     IS  [[pro1.global,RECEIVER],
                                      [pro2.global,RECEIVER],
                                        [con.global,SENDER]] :

  [SIZE chan.map]CHAN OF ANY  loc.chan :
  -- abbreviations for convenience only
  CHAN OF ANY         buff.in.one    IS  loc.chan[0] :
  CHAN OF ANY         buff.in.two    IS  loc.chan[1] :
  CHAN OF ANY         buff.out       IS  loc.chan[2] :

  PRI PAR
    csp.kernel(node.id, node.link[node.id],
               chan.map, loc.chan)
    buffer(buff.in.one, buff.in.two, buff.out)

:
-----------------------------------------------------------
-----------------------------------------------------------
PROC node3() --  procedure to run on Transputer 3
-----------------------------------------------------------

  #USE "\csplib\cspproc.tsr"   -- the interface library
  #USE "procs.tsr"             -- library of procedures
  #USE "aport.tsr"             -- apportionment structures

  VAL INT             node.id       IS  3 :
  VAL [][2]INT        chan.map      IS  [[con.global,RECEIVER]] :

  [SIZE chan.map]CHAN OF ANY  loc.chan :
  -- abbreviations for convenience only
  CHAN OF ANY         con.in        IS  loc.chan[0] :

  PRI PAR
    csp.kernel(node.id, node.link[node.id],
               chan.map, loc.chan)
    consumer(con.in)

:
-----------------------------------------------------------
```

LIST OF REFERENCES

[Br87] Bryant, Gregory R., *Transputer Instruction Set Disassembler*, August 1987.

[Ga86] Garret, D. R., *A Software System Implementation Guide and System Prototyping Facility for the MCORTEX Executive on the Real Time Cluster*, M.S. Thesis, December 1986, Naval Postgraduate School, Monterey, California.

[Ha87] Hart, Simon J., *Design, Implementation, and Evaluation Of A Virtual Shared Memory System In A Multi-Transputer Network*, M.S. Thesis, December 1987, Naval Postgraduate School, Monterey, California.

[Ho79] Hoare, C. A. R., "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666-677, August 1978.

[In86] *Product Information The Transputer Family*, March 1986, INMOS Ltd., Bristol, United Kingdom.

[In87a] *Transputer Development System 2.0, Programming Interface*, April 1987, INMOS Ltd., Bristol, United Kingdom.

[In87b] *Transputer Reference Manual*, January 1987, INMOS Ltd., Bristol, United Kingdom.

[In87c] *T800 Preliminary Data Sheet*, INMOS Ltd., 1987, Bristol, United Kingdom.

[In87d] *The Transputer Instruction Set—A Compiler Writer's Guide*, May 1987, INMOS Ltd., Bristol, United Kingdom.

[In87e] *T424 Transputer Reference Manual*, INMOS Ltd., 1987, Bristol, United Kingdom.

[Pe88] Discussion with Mr. Laurie Pegum, April 1988, INMOS Ltd., Colorado Springs, Colorado

[PoMa87] *A Tutorial Introduction to OCCAM Including Language Definition*, March 1987, INMOS LTD., Bristol, United Kingdom.

[Re85] *Ingres Reference Manual*, 1985, Relational Technology Inc., Alameda, California.

[ReKa79] Reed, D. P., and Kanodia, R. K., "Synchronization with Event Counts and Sequencers," *Communications of the ACM*, vol. 22, no. 2, pp. 115-123, February 1979.

[Va87] Vanni, Filho J., *Test and Evaluation of the Transputer in a Multi-Transputer Configuration*, M.S. Thesis, June 1987, Naval Postgraduate School, Monterey, California.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
4.	Dr. Uno R. Kodres, Code 52Kr Department of Computer Science Naval Postgraduate School Monterey, CA 93943	3
5.	Major Richard A. Adams, USAF, Code 52Ad Department of Computer Science Naval Postgraduate School Monterey, CA 93943	3
6.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
7.	Daniel Green, Code 20F Naval Surface Weapons Center Dahlgren, VA 22449	1
8.	Jerry Gaston, Code N24 Naval Surface Weapons Center Dahlgren, VA 22449	1
9.	Captain J. Hood, USN PMS 400B5 Naval Sea Systems Command Washington, DC 20362	1

10.	RCA AEGIS Repository RCA Corporation Government Systems Division Mail Stop 127-327 Moorestown, NJ 08057	1
11.	Library (Code E33-05) Naval Surface Weapons Center Dahlgren, VA 22449	1
12.	Dr. M. J. Gralia Applied Physics Laboratory Johns Hopkins Road Laurel, MD 20702	1
13.	Dana Small, Code 8242 Naval Ocean Systems Center San Diego, CA 92152	1
14.	Lieutenant Commander G. R. Bryant, USN Mare Island Naval Shipyard Vallejo, CA 94592	2
15.	Lieutenant Commander S. J. Hart Royal Australian Navy Combat Data Systems Centre 84 Maryborough Street Fyshwick, ACT 2610 AUSTRALIA	1
15.	AEGIS Modeling Laboratory, Code 52 Department of Computer Science Naval Postgraduate School Monterey, CA 93943	6
16.	Lieutenant W. R. Cloughley, USN 1448 47th Street Sacramento, CA 95819	1

317 592

Thesis
B82923 Bryant
c.1 Design, implementation and evaluation of an abstract programming and communications interface for a network of transputers.